КЛАССИКА COMPUTER SCIENCE

ВКЛЮЧАЯ C# 4.0

ЯЗЫК ПРОГРАММИРОВАНИЯ C#

ЧЕТВЕРТОЕ ИЗДАНИЕ



АНДЕРС ХЕЙЛСБЕРГ

М. ТОРГЕРСЕН, С. ВИЛТАМУТ, П. ГОЛД

Addison Wesley

ПИТЕР

Anders Hejlsberg, Mads Torgersen,
Scott Wiltamuth, Peter Golde

C#
Programming Language
(Covering C# 4.0)

4th Edition

Addison
Wesley

КЛАССИКА COMPUTER SCIENCE

А. Хейлсберг, М. Торгерсен,
С. Вилтамут, П. Голд

ЯЗЫК ПРОГРАММИРОВАНИЯ C#

ЧЕТВЕРТОЕ ИЗДАНИЕ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2012

ББК 32.973.2-018.1
УДК 004.43
Я41

Хейлсберг А., Торгерсен М., Вилтамут С., Голд П.

Я41 Язык программирования C#. Классика Computers Science. 4-е изд. — СПб.: Питер, 2012. — 784 с.: ил.

ISBN 978-5-459-00283-6

Перед вами — четвертое издание главной книги по языку C#, написанной легендой программирования Андерсом Хейлсбергом, архитектором C#, Delphi и Turbo Pascal, совместно с другими специалистами, входившими в группу разработчиков C# компании Microsoft. Издание является наиболее полным описанием языка и самым авторитетным источником информации по этой теме, построенным в формате сборника спецификаций, включающих в себя описание синтаксиса, сопутствующие материалы и примеры, а также образцы кода. Эта книга — своего рода «библия» разработчика, которая с легкостью может заменить как MSDN, так и остальные книги по C#.

Четвертое издание содержит описание новых особенностей C# 4.0, включая динамическое связывание, поименованные и необязательные параметры, а также ковариантные и контравариантные обобщенные типы. Цель этих новшеств — расширение возможностей C# для взаимодействия с объектами, не относящимися к платформе .NET. Отличительная особенность нового издания также состоит в том, что каждая глава книги содержит обширные комментарии, написанные известными «гуру» программирования, такими как Джон Скит, Джозеф Альбахари, Билл Вагнер, Кристиан Нейгел, Эрик Липперт и другие.

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-0321741769 англ.
ISBN 978-5-459-00283-6

Краткое оглавление

Предисловие . . . 21

К читателю . . . 23

Об авторах . . . 25

Об авторах комментариев . . . 26

Глава 1. Введение . . . 28

Глава 2. Лексическая структура . . . 83

Глава 3. Основные понятия . . . 113

Глава 4. Типы . . . 151

Глава 5. Переменные . . . 179

Глава 6. Приведение типов . . . 201

Глава 7. Выражения . . . 236

Глава 8. Операторы . . . 393

Глава 9. Пространства имен . . . 441

Глава 10. Классы . . . 454

Глава 11. Структуры . . . 589

Глава 12. Массивы . . . 607

Глава 13. Интерфейсы . . . 614

Глава 14. Перечисления . . . 642

Глава 15. Делегаты . . . 648

Глава 16. Исключения . . . 657

Глава 17. Атрибуты . . . 663

Глава 18. Небезопасный код . . . 683

Приложение А. Документирующие комментарии . . . 713

Приложение Б. Грамматика . . . 736

Приложение В. Ссылки . . . 773

Оглавление

Предисловие 21

К читателю 23

Об авторах 25

Об авторах комментариев 26

Глава 1. Введение 28

1.1. Здравствуй, Мир 30
1.2. Структура программы 31
1.3. Типы и переменные 33
1.4. Выражения 39
1.5. Операторы 41
1.6. Классы и объекты 45
1.6.1. Элементы класса 46
1.6.2. Доступность 47
1.6.3. Параметры-типы 48
1.6.4. Базовые классы 49
1.6.5. Поля 50
1.6.6. Методы 51
1.6.6.1. Параметры 52
1.6.6.2. Тело метода и локальные переменные 55
1.6.6.3. Статические методы и методы экземпляра 56
1.6.6.4. Виртуальные, переопределенные и абстрактные методы 57
1.6.6.5. Перегрузка методов 60
1.6.7. Другие функциональные элементы класса 62
1.6.7.1. Конструкторы 64
1.6.7.2. Свойства 64
1.6.7.3. Индексаторы 67
1.6.7.4. События 68
1.6.7.5. Операции 69
1.6.7.6. Деструкторы 70
1.7. Структуры 71
1.8. Массивы 73
1.9. Интерфейсы 75
1.10. Перечисления 77
1.11. Делегаты 79
1.12. Атрибуты 80

Глава 2. Лексическая структура 83

2.1. Программы 83
2.2. Грамматики 83
2.2.1. Система обозначений 83
2.2.2. Лексическая грамматика 85
2.2.3. Синтаксическая грамматика 85
2.3. Лексический анализ 85
2.3.1. Символы конца строки 86
2.3.2. Комментарии 86
2.3.3. Пробельные символы 88
2.4. Лексемы 88
2.4.1. Управляющие последовательности Unicode 88
2.4.2. Идентификаторы 89
2.4.3. Ключевые слова 92
2.4.4. Литералы (константы) 93
2.4.4.1. Булевские константы 93
2.4.4.2. Целочисленные константы 93
2.4.4.3. Вещественные константы 95
2.4.4.4. Символьные константы 96
2.4.4.5. Строковые константы 97
2.4.4.6. Константа null 100
2.4.5. Знаки операций и пунктуации 100
2.5. Директивы препроцессора 101
2.5.1. Символы условной компиляции 102
2.5.2. Выражения препроцессора 103
2.5.3. Директивы объявлений 103
2.5.4. Директивы условной компиляции 105
2.5.5. Директивы диагностики 108
2.5.6. Директивы region 109
2.5.7. Директивы line 109
2.5.8. Директива pragma 110
2.5.8.1. Предупреждения pragma warning 111

Глава 3. Основные понятия 113

3.1. Запуск приложения 113
3.2. Завершение работы приложения 114
3.3. Объявления 115
3.4. Элементы пространств имен и типов 118
3.4.1. Элементы пространств имен 119
3.4.2. Элементы структур 119
3.4.3. Элементы перечислений 120
3.4.4. Элементы классов 120
3.4.5. Элементы интерфейсов 120
3.4.6. Элементы массивов 120
3.4.7. Элементы делегатов 120
3.5. Доступ к элементам 121
3.5.1. Объявление вида доступа 121
3.5.2. Области доступности 124
3.5.3. Доступ protected для элементов экземпляров 127
3.5.4. Ограничения доступа 129

3.6. Сигнатуры и перегрузка . . . 130
3.7. Области видимости . . . 133
3.7.1. Скрытие имен . . . 136
3.7.1.1. Скрытие в результате вложения . . . 137
3.7.1.2. Скрытие в результате наследования . . . 138
3.8. Имена пространств имен и типов . . . 140
3.8.1. Полные имена . . . 143
3.9. Автоматическое управление памятью . . . 144
3.10. Порядок выполнения . . . 149

Глава 4. Типы . . . 151

4.1. Типы-значения . . . 151
4.1.1. Тип System.ValueType . . . 152
4.1.2. Конструкторы по умолчанию . . . 153
4.1.3. Структурные типы . . . 154
4.1.4. Простые типы . . . 155
4.1.5. Целочисленные типы . . . 156
4.1.6. Типы с плавающей точкой . . . 158
4.1.7. Десятичный тип . . . 161
4.1.8. Булевский тип . . . 162
4.1.9. Перечислимые типы . . . 163
4.1.10. Обнуляемые типы . . . 163
4.2. Ссылочные типы . . . 164
4.2.1. Классы . . . 165
4.2.2. Тип object . . . 166
4.2.3. Тип dynamic . . . 166
4.2.4. Тип string . . . 166
4.2.5. Интерфейсы . . . 166
4.2.6. Массивы . . . 166
4.2.7. Делегаты . . . 167
4.3. Упаковка и распаковка . . . 167
4.3.1. Упаковка . . . 168
4.3.2. Распаковка . . . 170
4.4. Сконструированные типы . . . 172
4.4.1. Аргументы-типы . . . 173
4.4.2. Открытые и закрытые типы . . . 173
4.4.3. Ограниченные и неограниченные типы . . . 174
4.4.4. Соблюдение ограничений . . . 174
4.5. Параметры-типы . . . 175
4.6. Типы деревьев выражений . . . 176
4.7. Тип dynamic . . . 177

Глава 5. Переменные . . . 179

5.1. Категории переменных . . . 179
5.1.1. Статические переменные . . . 180
5.1.2. Переменные экземпляра . . . 180
5.1.2.1. Переменные экземпляра в классах . . . 180
5.1.2.2. Переменные экземпляра в структурах . . . 181
5.1.3. Элементы массива . . . 181

5.1.4. Параметры-значения 181
5.1.5. Параметры-ссылки 182
5.1.6. Выходные параметры 183
5.1.7. Локальные переменные 183
5.2. Значение по умолчанию 185
5.3. Явное присваивание 186
5.3.1. Инициализированные переменные 187
5.3.2. Неинициализированные переменные 187
5.3.3. Точные правила для определения явного присваивания 188
5.3.3.1. Общие правила для операторов 188
5.3.3.2. Блок, операторы checked и unchecked 189
5.3.3.3. Оператор-выражение 189
5.3.3.4. Операторы объявления 189
5.3.3.5. Условный оператор 190
5.3.3.6. Оператор выбора 190
5.3.3.7. Оператор цикла while 190
5.3.3.8. Оператор цикла do 191
5.3.3.9. Оператор цикла for 191
5.3.3.10. Операторы перехода break, continue и goto 191
5.3.3.11. Оператор throw 191
5.3.3.12. Оператор return 192
5.3.3.13. Оператор try-catch 192
5.3.3.14. Оператор try-fnally 192
5.3.3.15. Оператор try-catch-finally 193
5.3.3.16. Оператор foreach 194
5.3.3.17. Оператор using 194
5.3.3.18. Оператор lock 194
5.3.3.19. Оператор yield 194
5.3.3.20. Общие правила для простых выражений 195
5.3.3.21. Общие правила для выражений с вложенными выражениями 195
5.3.3.22. Выражения вызова и выражения создания объекта 195
5.3.3.23. Выражения простого присваивания 196
5.3.3.24. &&-выражения 196
5.3.3.25. Выражения || 197
5.3.3.26. Выражения ! 198
5.3.3.27. Выражения ?? 198
5.3.3.28. Выражения ?: 198
5.3.3.29. Анонимные функции 199
5.4. Ссылки на переменные 200
5.5. Атомарность ссылок на переменные 200

Глава 6. Приведение типов 201

6.1. Неявные приведения типов 202
6.1.1. Тождественные преобразования 202
6.1.2. Неявные приведения арифметических типов 203
6.1.3. Неявные приведения перечислений 203
6.1.4. Неявные приведения обнуляемых типов 204
6.1.5. Неявные приведения литерала null 205

6.1.6. Неявные приведения ссылочных типов 205
6.1.7. Преобразования упаковки 207
6.1.8. Неявные приведения типа dynamic 207
6.1.9. Неявные приведения константных выражений 208
6.1.10. Неявные приведения параметров-типов 209
6.1.11. Определенные пользователем неявные приведения типов 210
6.1.12. Приведения анонимной функции и группы методов 210
6.2. Явные приведения типов 210
6.2.1. Явные приведения арифметических типов 210
6.2.2. Явные приведения перечислений 213
6.2.3. Явные приведения обнуляемых типов 213
6.2.4. Явные приведения ссылочных типов 214
6.2.5. Преобразование распаковки 215
6.2.6. Явные приведения типа dynamic 216
6.2.7. Явное приведение параметров-типов 217
6.2.8. Определенные пользователем явные приведения типов 218
6.3. Стандартные приведения типов 219
6.3.1. Стандартные неявные преобразования 219
6.3.2. Стандартные явные преобразования 219
6.4. Приведения типов, определенные пользователем 220
6.4.1. Допустимые приведения типов, определенные пользователем 220
6.4.2. Повышающие (lifted) приведения типов 220
6.4.3. Выполнение определенных пользователем приведений 221
6.4.4. Определенные пользователем неявные приведения типов 222
6.4.5. Определенные пользователем явные приведения типов 223
6.5. Приведения анонимных функций 225
6.5.1. Выполнение приведений анонимных функций к типам делегатов 227
6.5.2. Выполнение приведений анонимных функций к деревьям выражений 228
6.5.3. Пример реализации 228
6.6. Приведения групп методов 232

Глава 7. Выражения 236

7.1. Классификация выражений 236
7.1.1. Значение выражений 238
7.2. Статическое и динамическое связывание 239
7.2.1. Время связывания 240
7.2.2. Динамическое связывание 240
7.2.3. Типы составных частей выражений 242
7.3. Операции 243
7.3.1. Приоритет и ассоциативность операций 243
7.3.2. Перегрузка операций 245
7.3.3. Разрешение перегрузки унарных операций 247
7.3.4. Разрешение перегрузки бинарной операции 247
7.3.5. Набор определенных пользователем операций 248
7.3.6. Числовое расширение 248
7.3.6.1. Унарное числовое расширение 249
7.3.6.2. Бинарное числовое расширение 249
7.3.7. Расширенные операции 250

7.4. Поиск элемента 251
7.4.1. Базовые типы 254
7.5. Функциональные элементы 254
7.5.1. Списки аргументов. 257
7.5.1.1. Соответствующие параметры. 258
7.5.1.2. Вычисление списка аргументов во время выполнения программы 260
7.5.2. Выведение типов 262
7.5.2.1. Первая фаза. 264
7.5.2.2. Вторая фаза 265
7.5.2.3. Входные типы 265
7.5.2.4. Выходные типы 265
7.5.2.5. Зависимость 265
7.5.2.6. Выведение выходного типа 266
7.5.2.7. Явное выведение типа параметра 266
7.5.2.8. Точное выведение 266
7.5.2.9. Выведение нижней границы 267
7.5.2.10. Выведение верхней границы 268
7.5.2.11. Фиксация 269
7.5.2.12. Выведенный тип возвращаемого значения 270
7.5.2.13. Выведение типа для преобразования группы методов 271
7.5.2.14. Нахождение наилучшего общего типа для набора выражений 272
7.5.3. Разрешение перегрузки 272
7.5.3.1. Подходящий функциональный элемент .273
7.5.3.2. Лучший функциональный элемент 274
7.5.3.3. Лучшее приведение из выражения 276
7.5.3.4. Лучшее приведение из типа 276
7.5.3.5. Лучший целевой объект приведения 276
7.5.3.6. Перегрузка в обобщенных классах 277
7.5.4. Проверка динамического разрешения перегрузки во время компиляции. 277
7.5.5. Вызов функционального элемента 278
7.5.5.1. Вызовы для упакованных экземпляров .280
7.6. Первичные выражения 280
7.6.1. Литералы 281
7.6.2. Простые имена 281
7.6.2.1. Инвариантное значение в блоках 283
7.6.3. Выражения в скобках. 285
7.6.4. Доступ к элементу 285
7.6.4.1. Идентичные простые имена и имена типов 288
7.6.4.2. Грамматическая неоднозначность. 288
7.6.5. Выражения вызова. 289
7.6.5.1. Вызов метода 290
7.6.5.2. Вызов метода расширения 294
7.6.5.3. Вызов делегата 299
7.6.6. Доступ к элементу массива 299
7.6.6.1. Доступ к массиву. 300
7.6.6.2. Доступ к индексатору 301
7.6.7. this-доступ 302

7.6.8. Base-доступ . . . 303
7.6.9. Постфиксные инкрементные и декрементные операции . . . 304
7.6.10. Операция new . . . 306
7.6.10.1. Выражения создания объекта . . . 306
7.6.10.2. Инициализаторы объектов . . . 308
7.6.10.3. Инициализаторы коллекций . . . 311
7.6.10.4. Выражения создания массива . . . 313
7.6.10.5. Выражения создания делегата . . . 315
7.6.10.6. Выражение создания анонимного объекта . . . 317
7.6.11. Операция typeof . . . 319
7.6.12. Операции checked и unchecked . . . 322
7.6.13. Выражения значений по умолчанию . . . 325
7.6.14. Выражения анонимных методов . . . 325
7.7. Унарные операции . . . 325
7.7.1. Унарная операция плюс . . . 326
7.7.2. Унарная операция минус . . . 326
7.7.3. Логическая операция отрицания . . . 327
7.7.4. Операция поразрядного дополнения . . . 327
7.7.5. Префиксные инкрементные и декрементные операции . . . 328
7.7.6. Выражения приведения . . . 329
7.8. Арифметические операции . . . 331
7.8.1. Операция умножения . . . 331
7.8.2. Операция деления . . . 333
7.8.3. Операция взятия остатка от деления . . . 334
7.8.4. Операция сложения . . . 336
7.8.5. Операция вычитания . . . 339
7.9. Операции сдвига . . . 341
7.10. Операции отношения и операции проверки типа . . . 342
7.10.1. Операции сравнения для целочисленных типов . . . 343
7.10.2. Операции сравнения для типов с плавающей точкой . . . 344
7.10.3. Операции сравнения для десятичного типа . . . 346
7.10.4. Операции равенства для булевского типа . . . 346
7.10.5. Операции сравнения для перечислений . . . 346
7.10.6. Операции равенства для ссылочных типов . . . 347
7.10.7. Операции равенства для строкового типа . . . 349
7.10.8. Операции равенства для типов делегатов . . . 349
7.10.9. Операции равенства и константа null . . . 350
7.10.10. Операция is . . . 350
7.10.11. Операция as . . . 351
7.11. Логические операции . . . 353
7.11.1. Логические операции для целочисленных типов . . . 353
7.11.2. Логические операции для перечислений . . . 354
7.11.3. Булевские логические операции . . . 354
7.11.4. Обнуляемые булевские логические операции . . . 354
7.12. Условные логические операции . . . 355
7.12.1. Булевские условные логические операции . . . 356
7.12.2. Определенные пользователем условные логические операции . . . 356
7.13. Операция объединения с нулем . . . 357
7.14. Условная операция . . . 359
7.15. Выражения анонимных функций . . . 361

7.15.1. Сигнатуры анонимной функции 363
7.15.2. Тело анонимной функции 364
7.15.3. Разрешение перегрузки 365
7.15.4. Анонимные функции и динамическое связывание 366
7.15.5. Внешние переменные 366
7.15.5.1. Захваченные внешние переменные. 366
7.15.5.2. Инстанцирование локальных переменных 367
7.15.6. Вычисление выражений анонимных функций 370
7.16. Выражения запроса 370
7.16.1. Неоднозначность в выражениях запроса 372
7.16.2. Преобразование выражений запроса. 373
7.16.2.1. Конструкции select и GroupBy с продолжениями. 374
7.16.2.2. Явно определенные типы переменных диапазона 375
7.16.2.3. Вырожденные выражения запросов 376
7.16.2.4. Конструкции from, let, where, join и orderby. 376
7.16.2.5. Конструкции select 380
7.16.2.6. Конструкция GroupBy 380
7.16.2.7. Прозрачные идентификаторы. 380
7.16.3. Паттерн выражения запроса 382
7.17. Операции присваивания 384
7.17.1. Простое присваивание 385
7.17.2. Сложное присваиваивание. 388
7.17.3. Присваивание событий 389
7.18. Выражения 390
7.19. Константные выражения 390
7.20. Булевские выражения. 392

Глава 8. Операторы 393

8.1. Конечные точки и достижимость. 393
8.2. Блоки 396
8.2.1. Списки операторов 396
8.3. Пустой оператор 397
8.4. Помеченные операторы 399
8.5. Операторы объявления. 400
8.5.1. Объявление локальных переменных. 400
8.5.2. Объявление локальных констант 403
8.6. Операторы-выражения. 404
8.7. Операторы выбора 405
8.7.1. Оператор if. 405
8.7.2. Оператор switch 406
8.8. Операторы цикла 411
8.8.1. Оператор while 411
8.8.2. Оператор do 412
8.8.3. Оператор for 413
8.8.4. Оператор foreach 414
8.9. Операторы перехода. 420
8.9.1. Оператор break 421
8.9.2. Оператор continue 422
8.9.3. Оператор goto 423

8.9.4. Оператор return 425
8.9.5. Оператор throw 425
8.10. Оператор try 427
8.11. Операторы checked и unchecked 432
8.12. Оператор lock 432
8.13. Оператор using 434
8.14. Оператор yield 438

Глава 9. Пространства имен 441

9.1. Единица компиляции 441
9.2. Объявления пространств имен 442
9.3. Внешние псевдонимы 444
9.4. Директивы using 445
9.4.1. Using-директива псевдонима 445
9.4.2. Using-директива пространства имен 448
9.5. Элементы пространства имен 450
9.6. Объявление типов 450
9.7. Спецификатор псевдонима пространства имен 451
9.7.1. Уникальность псевдонимов 453

Глава 10. Классы 454

10.1. Объявления классов 454
10.1.1 Модификаторы класса 454
10.1.1.1. Абстрактные классы 455
10.1.1.2. Бесплодные классы 456
10.1.1.3. Статические классы 457
10.1.2. Модификатор partial 458
10.1.3. Параметры-типы 458
10.1.4. Спецификация базового класса 459
10.1.4.1. Базовые классы 459
10.1.4.2. Реализации интерфейсов 462
10.1.5. Ограничения на параметры-типы 462
10.1.6. Тело класса 468
10.2. Частичные типы 468
10.2.1. Атрибуты 469
10.2.2. Модификаторы 470
10.2.3. Параметры-типы и ограничения 470
10.2.4. Базовый класс 471
10.2.5. Базовые интерфейсы 471
10.2.6. Элементы 472
10.2.7. Частичные методы 473
10.2.8. Привязка имен 477
10.3. Элементы класса 478
10.3.1. Тип экземпляра 479
10.3.2. Элементы сконструированных типов 480
10.3.3. Наследование 482
10.3.4. Модификатор new 483
10.3.5. Модификаторы доступа 484
10.3.6. Составляющие типы 484

10.3.7. Статические элементы и элементы экземпляра 484
10.3.8. Вложенные типы 486
10.3.8.1. Полные имена 486
10.3.8.2. Объявления вида доступа 486
10.3.8.3. Скрытие 487
10.3.8.4. this-доступ 488
10.3.8.5. Доступ к закрытым и защищенным элементам охватывающего типа 489
10.3.8.6. Вложенные типы в обобщенных классах. 490
10.3.9. Зарезервированные имена элементов. 491
10.3.9.1. Имена элементов, зарезервированные для свойств 492
10.3.9.2. Имена элементов, зарезервированные для событий 493
10.3.9.3. Имена элементов, зарезервированные для индексаторов . . 493
10.3.9.4. Имена элементов, зарезервированные для деструкторов . . 493
10.4. Константы 493
10.5. Поля 496
10.5.1. Статические поля и поля экземпляра 497
10.5.2. Поля, доступные только для чтения 498
10.5.2.1. Использование статических полей, доступных только для чтения, вместо констант 499
10.5.2.2. Управление версиями констант и статических полей, доступных только для чтения 500
10.5.3. Асинхронно-изменяемые поля 501
10.5.4. Инициализация полей 503
10.5.5. Инициализаторы переменных 503
10.5.5.1. Инициализация статических полей 504
10.5.5.2. Инициализация полей экземпляра 506
10.6. Методы 506
10.6.1. Параметры метода. 509
10.6.1.1. Параметры-значения 511
10.6.1.2. Параметры-ссылки 511
10.6.1.3. Выходные параметры. 512
10.6.1.4. Параметры-массивы 514
10.6.2. Статические методы и методы экземпляра. 517
10.6.3. Виртуальные методы 517
10.6.4. Переопределенные методы 520
10.6.5. Бесплодные методы 523
10.6.6. Абстрактные методы 524
10.6.7. Внешние методы 526
10.6.8. Частичные методы 527
10.6.9. Методы расширения 527
10.6.10. Тело метода. 529
10.6.11. Перегрузка методов 530
10.7. Свойства 530
10.7.1. Статические свойства и свойства экземпляра 531
10.7.2. Коды доступа 532
10.7.3. Автоматически реализуемые свойства 538
10.7.4. Доступность 540
10.7.5. Виртуальные, бесплодные, переопределенные и абстрактные коды доступа. 541

10.8. События 544
10.8.1. События, подобные полям 546
10.8.2. Коды доступа событий 548
10.8.3. Статические события и события экземпляра 549
10.8.4. Виртуальные, бесплодные, переопределенные и абстрактные коды доступа 550
10.9. Индексаторы 551
10.9.1. Перегрузка индексаторов 555
10.10. Операции 555
10.10.1. Унарные операции 557
10.10.2. Бинарные операции 558
10.10.3. Операции преобразования 559
10.11. Конструкторы экземпляра 563
10.11.1. Инициализаторы конструктора 564
10.11.2. Инициализаторы переменных экземпляра 565
10.11.3. Выполнение конструктора 565
10.11.4. Конструкторы по умолчанию 568
10.11.5. Закрытые конструкторы 568
10.11.6. Необязательные параметры конструкторов экземпляра 569
10.12. Статические конструкторы 570
10.13. Деструкторы 573
10.14. Итераторы 575
10.14.1. Интерфейсы-перечислители 575
10.14.2. Перечислимые интерфейсы 576
10.14.3. Результирующий тип 576
10.14.4. Объекты-перечислители 576
10.14.4.1. Метод MoveNext 577
10.14.4.2. Свойство Current 579
10.14.4.3. Метод Dispose 579
10.14.5. Перечислимые объекты 579
10.14.5.1. Метод GetEnumerator 580
10.14.6. Пример реализации 580

Глава 11. Структуры 589

11.1. Объявления структур 590
11.1.1. Модификаторы структур 590
11.1.2. Модификатор partial 591
11.1.3. Интерфейсы структуры 591
11.1.4. Тело структуры 591
11.2. Элементы структуры 591
11.3. Различия между классами и структурами 592
11.3.1. Значимая семантика 592
11.3.2. Наследование 594
11.3.3. Присваивание 594
11.3.4. Значения по умолчанию 595
11.3.5. Упаковка и распаковка 595
11.3.6. Значение this 598
11.3.7. Инициализаторы полей 598

11.3.8. Конструкторы 599
11.3.9. Деструкторы 600
11.3.10. Статические конструкторы 600
11.4. Примеры структур 601
11.4.1. Целочисленный тип для базы данных 601
11.4.2. Тип boolean для базы данных 604

Глава 12. Массивы 607

12.1. Типы массивов 607
12.1.1. Тип System.Array 608
12.1.2. Массивы и обобщенный интерфейс IList 608
12.2. Создание массива 610
12.3. Доступ к элементам массива 610
12.4. Элементы типа массива 610
12.5. Ковариантность массивов 610
12.6. Инициализаторы массива 612

Глава 13. Интерфейсы 614

13.1. Объявления интерфейсов 614
13.1.1. Модификаторы интерфейса 615
13.1.2. Модификатор partial 615
13.1.3. Списки параметров вариантного типа 616
13.1.3.1. Безопасность вариантности 616
13.1.3.2. Преобразование вариантности 617
13.1.4. Базовые интерфейсы 618
13.1.5. Тело интерфейса 619
13.2. Элементы интерфейса 620
13.2.1. Методы интерфейса 621
13.2.2. Свойства интерфейса 622
13.2.3. События интерфейса 622
13.2.4. Индексаторы интерфейса 623
13.2.5. Доступ к элементам интерфейса 623
13.3. Полные имена элементов интерфейса 625
13.4. Реализации интерфейсов 626
13.4.1. Явные реализации элементов интерфейса 628
13.4.2. Уникальность реализованных интерфейсов 631
13.4.3. Реализация обобщенных методов 632
13.4.4. Сопоставление интерфейсов 633
13.4.5. Наследование реализаций интерфейсов 637
13.4.6. Повторная реализация интерфейса 638
13.4.7. Абстрактные классы и интерфейсы 640

Глава 14. Перечисления 642

14.1. Объявление перечисления 642
14.2. Модификаторы перечисления 643
14.3. Элементы перечисления 644
14.4. Тип System.Enum 646
14.5. Значения перечислений и операции 647

Глава 15. Делегаты 648

15.1. Объявления делегатов 649
15.2. Совместимость делегатов 652
15.3. Создание экземпляра делегата 652
15.4. Вызов делегата 653

Глава 16. Исключения 657

16.1. Причины исключений 658
16.2. Класс System.Exception 659
16.3. Как обрабатываются исключения 659
16.4. Распространенные классы исключений. 661

Глава 17. Атрибуты 663

17.1. Классы атрибутов 664
17.1.1. Использование атрибутов 664
17.1.2. Позиционные и именованные параметры 666
17.1.3. Типы параметров атрибута 667
17.2. Спецификация атрибута. 667
17.3. Экземпляры атрибутов 674
17.3.1. Компиляция атрибута 674
17.3.2. Получение экземпляра атрибута во время выполнения 674
17.4. Зарезервированные атрибуты 675
17.4.1. Атрибут AttributeUsage. 675
17.4.2. Атрибут Conditional. 676
17.4.2.1. Условные методы 676
17.4.2.2. Классы условных атрибутов 679
17.4.3. Атрибут Obsolete. 680
17.5. Атрибуты для взаимодействия 682
17.5.1. Взаимодействие с компонентами COM и Win32 682
17.5.2. Взаимодействие с другими языками .NET. 682
17.5.2.1. Атрибут IndexerName 682

Глава 18. Небезопасный код 683

18.1. Небезопасные контексты 684
18.2. Типы указателей. 686
18.3. Фиксированные и перемещаемые переменные 690
18.4. Преобразования указателей 690
18.4.1. Массивы указателей 692
18.5. Использование указателей в выражениях. 693
18.5.1. Разыменование указателей 694
18.5.2. Доступ к элементу объекта через указатель 694
18.5.3. Доступ к элементу через указатель. 696
18.5.4. Операция получения адреса 696
18.5.5. Инкремент и декремент указателей 698
18.5.6. Арифметика указателей 698
18.5.7. Сравнение указателей. 699
18.5.8. Операция sizeof 699
18.6. Оператор fixed 700

18.7. Буферы фиксированного размера 705
18.7.1. Объявления буферов фиксированного размера. 706
18.7.2. Использование буферов фиксированного размера в выражениях 707
18.7.3. Проверка явного присваивания 708
18.8. Выделение памяти в стеке 709
18.9. Динамическое выделение памяти 710

Приложение А. Документирующие комментарии 713

А.1. Введение. 713
А.2. Рекомендуемые теги. 715
А.2.1. <c> 715
А.2.2. <code> 716
А.2.3. <example> 716
А.2.4. <exception> 716
А.2.5. <include> 717
А.2.6. <list> 718
А.2.7. <para> 719
А.2.8. <param> 719
А.2.9. <paramref>. 720
А.2.10. <permission> 720
А.2.11. <remark> 721
А.2.12. <returns> 721
А.2.13. <see> 721
А.2.14. <seealso>. 722
А.2.15. <summary> 722
А.2.16. <value>. 723
А.2.17. <typeparam> 723
А.2.18. <typeparamref> 723
А.3. Обработка файла документации 724
А.3.1. Формат ID-строки. 724
А.3.2. Примеры ID-строк 725
А.4. Пример 729
А.4.1. Исходный код на C# 729
А.4.2. Результирующий XML 732

Приложение Б. Грамматика 736

В.1. Лексическая грамматика 736
Б.1.1. Символы конца строки 736
Б.1.2. Комментарии 736
Б.1.3. Пробельные символы 737
Б.1.4. Лексемы. 737
Б.1.5. Управляющие последовательности Unicode 738
Б.1.6. Идентификаторы 738
Б.1.7. Ключевые слова 739
Б.1.8. Литералы 739
Б.1.9. Знаки операций и пунктуации 741
Б.1.10. Директивы препроцессора 741
Б.2. Синтаксическая грамматика 744
Б.2.1. Основные понятия 744

Б.2.2. Типы 744
Б.2.3. Переменные 746
Б.2.4. Выражения 746
Б.2.5. Операторы. 753
Б.2.6. Пространства имен 757
Б.2.7. Классы 758
Б.2.8. Структуры 765
Б.2.9. Массивы 766
Б.2.10. Интерфейсы 766
Б.2.11. Перечисления. 767
Б.2.12. Делегаты 768
Б.2.13. Атрибуты 768
Б.3. Расширения грамматики для небезопасного кода 770

Приложение В. Ссылки 773

Предисловие

Прошло десять лет с момента выпуска .NET летом 2000 года. Для меня значение .NET заключалось в применении управляемого кода для локальных данных и обмена данными между программами в XML-формате. Что было мне в то время неясно, так это то, насколько важное место впоследствии займет C#.

С самого начала развития .NET C# являлся главным инструментом, с помощью которого разработчики могли понимать платформу .NET и взаимодействовать с ней. Спросите любого разработчика .NET о различии между типом «значение» и типом «ссылка», и он или она тут же ответит: «Структура и класс», а не «Типы, производные и не производные от System.ValueType». Почему? Потому, что люди пользуются языками — а не API — для того, чтобы сообщать свои идеи и намерения среде выполнения и, что еще более важно, сообщать их друг другу.

Трудно переоценить значение хорошего языка для успешного развития платформы в целом. C# изначально был важен для определения направления, в котором люди будут думать о развитии .NET. Может быть, даже более важно, что в процессе развития .NET такие инструменты, как итераторы и замыкания (известные также как анонимные методы), были представлены разработчикам как черты, присущие языку, которые реализуются компилятором C#, а не как свойства самой платформы. Тот факт, что C# является основным центром инноваций для .NET, стал даже более очевидным с возникновением C# 3.0, с введением стандартизованных операций запроса, компактных лямбда-выражений, методов расширения и доступа к деревьям выражений во время выполнения — и опять все идет от развития языка и компилятора. Наиболее существенная особенность C# 4.0, динамический вызов, также скорее свойство языка и компилятора — в большей мере, чем следствие изменений в общеязыковой среде выполнения (Common Language Runtime, CLR).

Трудно говорить о C# и не упомянуть о человеке, который его изобрел и постоянно развивает, об Андерсе Хейлсберге. Особенно приятно было принимать участие в постоянных конференциях, посвященных разработке C# 3.0, проходивших в течение нескольких месяцев, во время которых можно было наблюдать его в процессе работы. У него удивительное чутье на то, что нравится или, наоборот, не нравится разработчикам — и в то же время он стремится в полной мере использовать возможности своей команды для достижения наилучшего результата.

Андерс продемонстрировал невероятную способность воспринимать ключевые идеи программистского сообщества и делать их доступными широкой аудитории. В особенности эта его способность проявилась при разработке C# 3.0. Это непростое искусство. Гай Стил однажды сказал о Java: «Мы не прочь превзойти разработчиков Lisp; мы следуем за программистами на C++. Мы помогли многим из них пройти полдороги к Lisp». Глядя на C# 3.0, я думаю, что C# помог пройти большую часть этого пути как минимум одному разработчику (мне). C# 4.0 делает следующий шаг по направлению к Lisp (а также к JavaScript, Python, Ruby и т. д.),

давая возможность создавать понятные программы, не основанные на статическом определении типов.

Людям нужен как C#, так и руководство, написанное на родном языке и использующее некий формализм (BNF), чтобы глубже разобраться в тонкостях и быть уверенными, что мы пишем на одном и том же C#. Книга, которую вы держите в руках, как раз и является таким руководством. Опираясь на собственный опыт, я могу с уверенностью утверждать, что каждый разработчик .NET, который прочтет его, откроет для себя что-то такое, что поможет ему выйти на новый уровень.

Наслаждайтесь!

Дон Бокс
Редмонд, Вашингтон, май 2010

К читателю

Проект C# начался более чем 12 лет назад, в декабре 1998 года и имел целью создание простого, современного, объектно-ориентированного и обеспечивающего безопасность типов языка программирования для новой платформы .NET, которая тогда еще не получила это название. С тех пор C# прошел большой путь. Этим языком теперь пользуются более миллиона программистов. Выпущено уже четыре версии, и в каждой добавлено несколько новых важных свойств.

Соответственно, это четвертое издание данной книги. В ней вы найдете полное техническое описание языка C#. Это последнее издание включает в себя две разновидности нового материала, отсутствующего в предыдущих версиях. Самым замечательным, конечно, является описание новых особенностей C# 4.0, включая динамическое связывание, именованные и необязательные параметры, а также ковариантные и контравариантные обобщенные типы. Главной задачей этих новшеств было расширение возможностей C# для взаимодействия с объектами, не относящимися к платформе .NET. Подобно тому как LINQ позволял ощутить возможности языковой интеграции, необходимой для создания кода, обеспечивающего доступ к внешним данным, динамическое связывание в C# 4.0 дает возможность естественным для C# образом взаимодействовать с объектами таких динамических языков программирования, как Python, Ruby и JavaScript.

Предыдущее издание книги содержало мнения широко известных экспертов в области C#. Это очень понравилось нашим читателям, и нам особенно приятно и в этом издании предложить вашему вниманию новый ряд глубоких и интересных комментариев как от прежних, так и от новых авторов по поводу основных принципов, взаимосвязей и перспектив языка. Отрадно сознавать, что эти отклики дополняют основные материалы и помогают живее воспринимать основные особенности C#.

В создании языка C# принимали участие многие люди. Команда разработчиков C# 1.0 состояла из Андерса Хейлсберга (Anders Hejlsberg), Скотта Вилтамута (Scott Wiltamuth), Питера Голда (Peter Golde), Питера Соллиша (Peter Golde) и Эрика Гуннерсона (Eric Gunnerson). Создатели версии C# 2.0: Андерс Хейлсберг, Питер Голд, Питер Халлам (Peter Hallam), Шон Катценбергер (Shon Katzenberger), Тодд Проэбстин (Todd Proebsting) и Ансон Хортон (Anson Horton).

Далее, замысел и способ реализации обобщений в C# и в среде выполнения .NET основаны на прототипе Gyro, разработанном Доном Саймом (Don Syme) и Эндрю Кеннеди (Andrew Kennedy) из Microsoft Research. Команда создателей версии C# 3.0: Андерс Хейлсберг, Эрик Мейер (Erik Meijer), Мэтт Уоррен (Matt Warren), Мадс Торгерсен (Mads Torgersen), Питер Халлам и Динеш Кулкарни (Dinesh Kulkarni). В разработке версии C# 4.0 принимали участие Андерс Хейлсберг, Мэтт Уоррен, Мадс Торгерсен, Эрик Липперт (Eric Lippert), Джим Хугунин (Jim Hugunin), Люциан Висчик (Lucian Wischik) и Нил Гафтер (Neal Gafter).

Здесь невозможно упомянуть всех людей, которые так или иначе повлияли на разработку C#, но тем не менее мы благодарим их всех. В вакууме нельзя создать ничего достойного; поэтому постоянная обратная связь, которую мы получаем от сообщества разработчиков, бесценна.

C# был и продолжает оставаться одним из наиболее привлекательных и вдохновляющих проектов, над которыми мы работаем. Мы надеемся, что вы получите удовольствие от знакомства с C#, так же как мы получали удовольствие, создавая его.

Андерс Хейлсберг
Мадс Торгерсен
Скотт Вилтамут
Питер Голд
Сиэтл, Вашингтон, сентябрь 2010

Об авторах

Андерс Хейлсберг — легенда программирования. Является архитектором языка C# и Microsoft Technical Fellow[1]. Вступил в корпорацию Microsoft в 1996 году после тринадцатилетней карьеры в компании Borland, где был главным архитектором Delphi и Turbo Pascal.

Мадс Торгерсен является руководителем группы разработчиков языка C# в корпорации Microsoft; ежедневно участвует в процессе создания языка и поддержке его спецификации.

Скотт Вилтамут — руководитель разработчиков Visual Studio Team Professional в корпорации Microsoft. Работал с широким спектром инструментов разработки, включая OLE Automation, Visual Basic, Visual Basic for Applications, VBScript, JScript, Visual J++ и Visual C#.

Питер Голд — ведущий разработчик компилятора C# компании Microsoft. В качестве основного представителя Microsoft в Европейской ассоциации производителей компьютеров (European Computer Manufactures Association, ECMA), которая стандартизовала C#, он руководит созданием компилятора и работает над проектированием языка. В настоящее время является архитектором компиляторов в корпорации Microsoft.

[1] Высшее звание в технологической иерархии Microsoft, человек, определяющий стратегию развития. — *Примеч. науч. ред.*

Об авторах комментариев

Брэд Абрамс (Brad Abrams) является одним из основных участников как команды создателей общеязыковой среды выполнения (CRL), так и команды, разрабатывающей .NET Framework в корпорации Microsoft, где он до недавнего времени был руководителем разработчиков WCF и WF. Брэд принимает участие в разработке .NET Framework с 1998 года, когда он начинал свою карьеру создателя каркаса, разрабатывая библиотеки базовых классов (Base Class Library, BCL) — ядро каркаса .NET. Брэд закончил университет штата Северная Каролина в 1997 году со степенью бакалавра в области компьютерных технологий. У него имеются следующие публикации: *Framework Design Guidelines*, *Second Edition* (Addison-Wesley, 2009) и *.NET Framework Standard Library Annotated Reference* (Volumes 1 and 2) (Addison-Wesley, 2006).

Джозеф Альбахари (Joseph Albahari) является соавтором книг *C# 4.0 in a Nutshell* (O'Reilly, 2007), *C# 3.0 Pocket Reference*(O'Reilly, 2008) и *LINQ Pocket Reference* (O'Reilly, 2008). Имеет семнадцатилетний опыт работы в качестве главного разработчика и архитектора программного обеспечения в области здравоохранения, образования и телекоммуникаций. Он также является автором LINQPad, утилиты для интерактивных запросов к базам данных в LINQ.

Кшиштоф Цвалина (Krzysztof Cwalina) — главный архитектор команды разработки каркаса .NET Framework в Microsoft. В начале карьеры разрабатывал программные интерфейсы приложений (Application Program Interface, API) для первого выпуска каркаса. Он постоянно участвует в разработке, продвижении и внедрении общих и архитектурных стандартов в процессе развития .NET Framework. Он является соавтором книги *FrameworkDesign Guidelines* (Addison-Wesley, 2005). Посетите его блог на http://blogs.msdn.com/kcwalina.

Джесс Либерти (Jesse Liberty) (ник Silverlight Geek) — старший разработчик в компании Microsoft, автор многочисленных бестселлеров для программистов и множества популярных статей. Он также любит поговорить о событиях мирового масштаба. Его блог http://JesseLiberty.com обязательно должны читать разработчики под WPF, Windows Phone 7 и Silverlight. У Джесса за плечами опыт программирования в течение двух десятков лет, в том числе на посту вице-президента Citi и в качестве известного разработчика в AT&T. Он доступен в своем блоге и в Твиттере как @JesseLiberty.

Эрик Липперт (Eric Lippert) — старший разработчик команды, создававшей компилятор C# в компании Microsoft. Он разрабатывал и реализовывал такие языки, как Visual Basic, VBScript, JScript, C#, а также Visual Studio Tools For Office. Обсуждение всех этих тем вы можете найти в его блоге на странице http://blogs.msdn.com/EricLippert.

Кристиан Нейгел (Christian Nagel) является региональным директором Microsoft и MVP. Он автор нескольких книг, в том числе *Professional C# 4 with .NET 4* (Wrox, 2010) и *Enterprise Services with the.NET Framework* (Addison-Wes-

ley, 2005). Как владелец CN innovation и партнер thinktecture, он обучает разработчиков различным технологиям Microsoft .NET. Кристиан доступен по адресу http://www.cninnovation.com.

Владимир Решетников (Vladimir Reshetnikov) — Microsoft MVP по Visual C#. Он более восьми лет занимается разработкой программного обеспечения и около шести лет посвятил развитию Microsoft .NET и C#. Его блог находится на странице http://nikov-thoughts.blogspot.com.

Марек Сафар (Marek Safar) — ведущий разработчик команды создателей компилятора Novell C#. Последние пять лет занимается разработкой основных свойств компилятора Mono C#. Посетите его блог на странице http://mareksafar.blogspot.com.

Крис Селлз (Chris Sells) является руководителем разработчиков в отделе Business Platform (синоним отдела SQL Server) корпорации Microsoft. Он написал несколько книг, в том числе *Programming WPF* (O'Reilly, 2007), *Windows Forms 2.0 Programming* (Addison-Wesley, 2006) и *ATL Internals* (Addison-Wesley, 1999). В свободное время Крис играет ведущую роль в различных конференциях и занимает видное место в списке участников внутренних обсуждений продуктов Microsoft. Более полную информацию о Крисе и его различных проектах можно найти на странице http://www.sellsbrothers.com.

Питер Сестофт (Peter Sestoft) — профессор информационных технологий университета Копенгагена, Дания, в области разработки программного обеспечения. Являлся членом Международного комитета по стандартизации C# Европейской ассоциации производителей компьютеров (Europian Computer Manufacturers Association, ECMA) с 2003 по 2006 год. Он также является автором книг *C# Precisely* (MIT Press, 2004) *и Java Precisely* (MIT Press, 2005). Его можно найти по адресу http://www.itu.dk/people/sestoft.

Джон Скит (Jon Skeet) — автор книги *C# in Depth* (Manning, 2010) и C# MVP. Он работает в Лондоне для компании Google, в свободное время говорит и пишет о C#. Его блог находится по адресу http://msmvps.com/jon.skeet или же вы можете найти его в рубрике ответов на вопросы на Stack Overfow (http://stackoverfow.com).

Билл Вагнер (Bill Wagner) — основатель компании SRT Solutions, региональный директор Microsoft и C# MVP. Провел большую часть своей профессиональной карьеры между фигурными скобками. Является автором книг *Effective C#* (Addison-Wesley, 2005) и *More Effective C#* (Addison-Wesley, 2009). Он также вел колонку, посвященную C#, в *Visual Studio Magazine* и внес большой вклад в развитие Центра разработчиков C# (C# Developer Center) при Microsoft Developer Network (MSDN). С развитием его идей о C# и другими темами можно ознакомиться на странице http://srtsolutions.com/blogs/billwagner.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1

Введение

C# (произносится Си шарп) — простой, современный, объектно-ориентированный, обеспечивающий безопасность типов язык программирования. C# происходит из семейства языков C и может быть легко освоен программирующими на C, C++ и Java. Он соответствует международному стандарту Европейской ассоциации производителей компьютеров (European Computer Manufacturers Association, ECMA) — стандарт **ECMA-334**, а также стандарту Международной организации по стандартизации (International Standards Organization, ISO) и Международной электротехнической комиссии (International Electrotechnical Commission, IEC) — стандарт **ISO/IEC 23270**. Компилятор Microsoft C# для .NET Framework согласуется с обоими этими стандартами.

Являясь объектно-ориентированным языком, C# в то же время обеспечивает поддержку компонентно-ориентированного программирования. Структура современного программного обеспечения все в большей мере основывается на независимых модулях, содержащих свое описание. Суть этих компонентов в том, что они являются частью модели программирования, основанной на свойствах, методах и событиях, они имеют атрибуты, обеспечивающие описательную информацию о компонентах, и они самодокументированы. C# обеспечивает языковые конструкции, непосредственно поддерживающие эти концепции, что делает его очень естественным языком для создания и применения компонентов программного обеспечения.

Свойства C#, помогающие создавать надежные и устойчивые приложения: *сборка мусора* автоматически освобождает память, занятую неиспользуемыми объектами; *обработка исключительных ситуаций* обеспечивает структурированный и расширяемый подход к обнаружению ошибок и восстановлению; структура языка, обеспечивающая безопасность типов, делает невозможным получать значения неинициализированных переменных, индексировать массивы вне их границ или выполнять бесконтрольное приведение типов.

C# имеет *единую систему типов*. Все типы, используемые в C#, в том числе примитивные, такие как int и double, происходят от единого корневого типа object. Таким образом, во всех типах могут использоваться общие для всех операции; значения любого типа хранятся, передаются и используются единообразно. Более того, C# поддерживает определяемые пользователем ссылочные типы и типы-значения, что позволяет осуществлять динамическое распределение памяти под объекты наряду с компактным хранением небольших структур в стеке.

Для того чтобы иметь уверенность в том, что программы и библиотеки последующих версий C# совместимы с предыдущими, особое внимание уделяется

управлению версиями C#. Во многих языках программирования уделяется мало внимания этой проблеме. В результате программы, написанные на этих языках, часто ломаются при появлении новых версий библиотек, используемых этими программами. Аспекты C#, непосредственно поддерживающие версионирование, включают в себя отдельные модификаторы `virtual` и `override`, правила разрешения перегрузки методов и поддержку явных описаний элементов интерфейса.

Остальная часть этой главы описывает ключевые особенности языка C#. В следующих главах мы опишем правила и исключения детально и временами будем обращаться к математическим выкладкам, а здесь мы стремимся изложить суть кратко и ясно, сознательно жертвуя полнотой описания. В данном случае целью является ознакомление читателя с основами языка, которое может облегчить написание простых программ и знакомство с последующим материалом.

КРИС СЕЛЛЗ

Я абсолютно согласен с тем, что C# является «современным, объектно-ориентированными и обеспечивающим безопасность типов», однако это далеко не все. Новые функциональные возможности языка, такие как обобщенные типы и анонимные делегаты в C# 2.0, LINQ в версии 3.0 и динамические значения в C# 4.0, позволили достичь того, что программы стали проще, их стало легче читать и поддерживать — что является целью любого языка программирования.

ЭРИК ЛИППЕРТ

C# также является языком с постоянно возрастающими функциональными возможностями. Такие свойства, как вывод типов, лямбда-выражения и «разумные» запросы в стиле монад, позволят разработчикам традиционных объектно-ориентированных программ использовать эти новые идеи для того, чтобы увеличить выразительные возможности языка.

КРИСТИАН НЕЙГЕЛ

C# не является объектно-ориентированным языком в чистом виде: скорее это язык, возможности которого постоянно расширяются, для того чтобы обеспечить наибольшую продуктивность в главных областях его применения. Программы, написанные на C# 3.0 с использованием конструкций функционального программирования, могут выглядеть совершенно иначе, чем программы, написанные на C# 1.0.

ДЖОН СКИТ

Некоторые черты C# определенно способствовали тому, что с течением времени он стал более функциональным — в то же время в версии C# 3.0 автоматически реализуемые свойства, так же как и инициализаторы объектов, способствовали изменяемости (mutability). Интересно будет увидеть, какие свойства в последующих версиях помогут достигнуть неизменяемости (immutability) наряду с поддержкой таких областей, как кортежи, сравнение с образцом (паттерн-матчинг) и хвостовая рекурсия.

БИЛЛ ВАГНЕР

Эта глава не изменилась с первой версии спецификации C#. Язык явно разросся и прибавились новые идиомы — и все же C# до сих пор является доступным для освоения. Новые функциональные возможности доступны, но не все они требуются для каждой программы. C# все еще остается подходящим и для неопытных разработчиков, хотя он становится все более мощным.

1.1. Здравствуй, Мир

Программа «Здравствуй, Мир» традиционно используется для введения в язык программирования. На языке C# эта программа выглядит следующим образом:

```
using System;
class Hello
{
    static void Main() {
        Console.WriteLine("Здравствуй, Мир!");
    }
}
```

Исходные файлы C# обычно имеют расширение `.cs`. Если программа «Здравствуй, Мир» хранится в файле `hello.cs`, ее можно скомпилировать компилятором Microsoft C# с помощью командной строки:

```
csc hello.cs
```

которая создает исполняемую сборку `hello.exe`. Это приложение выведет

```
Здравствуй, Мир!
```

Программа «Здравствуй, Мир» начинается с директивы `using`, которая ссылается на пространство имен `System`. Пространства имен представляют собой средства организации иерархий в программах и библиотеках C#. Пространства имен содержат типы и другие пространства имен — например, пространство `System` содержит некоторое количество типов, таких как класс `Console`, на который ссылается программа, и другие пространства имен, например `IO` и `Collections`. Директива `using`, ссылающаяся на данное пространство, позволяет использовать типы, которые в него входят, без уточняющих квалификаторов. Благодаря присутствию директивы `using` в программе можно использовать сокращенную запись `Console.WriteLine` вместо полной `System.Console.WriteLine`.

Класс `Hello`, объявленный в программе «Здравствуй, Мир», состоит из единственного элемента — метода `Main`. Метод `Main` объявлен с модификатором `static`. Методы экземпляра ссылаются на определенный экземпляр объекта, используя ключевое слово `this`, а статические методы не ссылаются на конкретный объект. По соглашению статический метод `Main` служит точкой входа программы.

Программа выполняет вывод на консоль с помощью метода `WriteLine` класса `Console` из пространства имен `System`. Этот класс входит в библиотеки классов .NET Framework, на которые по умолчанию автоматически ссылается

компилятор Microsoft C#. Заметим, что C# не имеет отдельной библиотеки, .NET Framework *и есть* та библиотека, к которой обращается C# во время работы программы.

> **БРЭД АБРАМС**
>
> Интересно отметить, что `Console.WriteLine()` — просто сокращение `Console.Out.WriteLine`. Здесь `Console.Out` — свойство, которое возвращает реализацию базового класса `System.IO.TextWriter`, предназначенного для выполнения вывода на консоль. Предыдущий пример может быть также корректно записан следующим образом:
>
> ```
> using System;
> class Hello
> {
>     static void Main() {
>         Console.Out.WriteLine(«Hello, World»);
>     }
> }
> ```
>
> При проектировании каркаса мы с самого начала обращали особое внимание на то, чтобы этот раздел спецификации языка C# был написан без лишних сложностей. Мы удобным способом оптимизировали перегрузку для `Console`, чтобы сделать «Здравствуй, Мир» еще проще для написания. По общим отзывам кажется, что это имело смысл. Сейчас вы почти не встретите вызовов `Console.Out.WriteLine()`.

1.2. Структура программы

Ключевые организующие концепции C# — это *программы, пространства имен, типы, элементы* и *сборки*. Программа на C# содержит один или более исходных файлов. В программах объявляются типы, которые содержат элементы и могут быть организованы в пространства имен. *Классы* и *интерфейсы* являются примерами типов. *Поля, методы, свойства* и *события* — примеры элементов. В процессе компиляции программы на C# они физически упаковываются в сборки. Сборки обычно имеют расширение `.exe` или `.dll`, в зависимости от того, выполняют они роль *приложения* или *библиотеки*.

Пример:

```
using System;
namespace Acme.Collections
    {
        public class Stack
        {
            Entry top;
            public void Push(object data) {
                top = new Entry(top, data);
            }
            public object Pop() {
```

продолжение ⇗

```
                if (top == null) throw new InvalidOperationException();
                object result = top.data;
                top = top.next;
                return result;
            }
            class Entry
            {
                public Entry next;
                public object data;
                public Entry(Entry next, object data) {
                    this.next = next;
                    this.data = data;
                }
            }
        }
    }
```

Здесь класс с именем `Stack` объявлен в пространстве имен `Acme.Collections`. Полное имя этого класса — `Acme.Collections.Stack`. Он содержит несколько элементов: поле с именем `top`, два метода — `Push` и `Pop`, и вложенный класс, именуемый `Entry`. Класс `Entry`, в свою очередь, включает в себя три элемента: поле `next`, поле `data` и конструктор. Предположим, что исходный код содержится в файле `acme.cs`. Командная строка

```
csc /t:library acme.cs
```

компилирует данный пример как библиотеку (код не содержит точку входа `Main`) и в результате создает сборку, названную `acme.dll`.

Сборки содержат исполняемый код в форме инструкций *промежуточного языка* (*Intermediate Language, IL*) и описательную информацию в форме *метаданных*. Перед исполнением IL-код сборки автоматически преобразуется в код, предназначенный для процессора, с помощью JIT-компилятора (Just-In-Time[1]) общеязыковой среды выполнения (CLR).

Поскольку сборка является функциональной единицей, содержащей собственное описание (код и метаданные), в C# нет необходимости использовать директивы `#include` и заголовочные файлы. Доступ к открытым типам и элементам, содержащимся в отдельной сборке в программе C#, обеспечивается просто ссылкой на эту сборку при компиляции программы. Например, приведенная далее программа использует класс `Acme.Collections.Stack` из сборки `acme.dll`:

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
```

[1] Переводится как «вовремя», то есть компилируются только те части программы, которые требуются в данный момент. — *Примеч. перев.*

```
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

Если программа хранится в файле `test.cs`, то при его компиляции на `acme.dll` можно ссылаться с помощью ключа компилятора `/r`:

```
csc /r:acme.dll test.cs
```

Таким образом создается исполняемая сборка `test.exe`, ее результат работы

```
100
10
1
```

C# позволяет хранить исходный текст программы в нескольких исходных файлах. При компиляции программы на C#, состоящей из нескольких файлов, все они обрабатываются вместе и могут свободно ссылаться один на другой — концептуально это то же самое, как если бы перед обработкой все исходные файлы были объединены в один большой файл. Предварительные описания в C# не нужны, поскольку, за редким исключением, порядок описаний не важен. В C# не требуется, чтобы в исходном файле описывался только один открытый тип или чтобы имя исходного файла соответствовало типу, объявленному в нем.

ЭРИК ЛИППЕРТ

Это свойство не похоже на язык Java. То, что в C# порядок объявлений не имеет значения, отличает C# и от C++.

КРИС СЕЛЛЗ

Заметим, что в предыдущем примере использован оператор `using Acme.Collections`, который выглядит как директива `#include` в стиле C, но это не так. Скорее это удобный способ присвоения имени, чтобы когда компилятор встречает `Stack`, он имел информацию о пространствах имен, в которых следует искать этот класс. Компилятор выполнял бы те же действия, если бы в примере использовалось полное имя:

```
Acme.Collections.Stack s = new Acme.Collections.Stack();
```

1.3. Типы и переменные

В C# существует две разновидности типов переменных: *типы-значения* и *ссылочные типы*. Переменные типа-значения содержат значения, а в переменных ссылочного типа (объектах) хранятся ссылки на данные. Для ссылочного типа переменных допустимо, чтобы две переменные ссылались на один объект, и, таким образом, действия над одной переменной влияют на объект, на который ссылается другая переменная. В случае типа-значения каждая переменная имеет собственную копию данных, и операция над одной переменной не может повлиять на другую (за исключением случая использования параметров `ref` и `out`).

ДЖОН СКИТ

Возможно, слово «ссылочный» (reference) для обозначения ссылочного типа переменных выбрано неудачно. Это приводит к огромной неразберихе (или по меньшей мере недопониманию) при рассмотрении различий в семантике передачи параметра по ссылке и передачи параметра по значению.

Различие между типом-значением и ссылочным типом, возможно, наиболее важный пункт для начинающих изучать C#: пока он не будет осознан в полной мере, ни о чем другом говорить не имеет смысла.

ЭРИК ЛИППЕРТ

Возможно, наиболее распространенное заблуждение относительно типа-значения состоит в том, что переменные этого типа «хранятся в стеке», в то время как переменные ссылочного типа «хранятся в куче». Во-первых, эта деталь относится к реализации исполнительной среды, а не к особенностям языка. Во-вторых, это объяснение не прибавляет ничего нового. В-третьих, это неправда: в самом деле, данные, ассоциируемые с экземплярами ссылочного типа, хранятся в куче, но эти данные могут включать в себя экземпляры типов-значений, и, таким образом, типы-значения иногда тоже хранятся в куче. В-четвертых, если бы различие между типом-значением и ссылочным типом заключалось в деталях хранения, то команда, создававшая CLR, назвала бы их «типом стека» и «типом кучи». Действительное различие состоит в том, что типы-значения копируются по значению, а ссылочные типы копируются по ссылке; каким образом среда выполнения выделяет память для обеспечения правил времени жизни, не столь важно для подавляющего большинства основных программных сценариев.

БИЛЛ ВАГНЕР

Программируя на C#, вы вынуждены принимать важное решение по выбору типов с учетом значимой и ссылочной семантики. Разработчики, использующие ваш тип данных, не могут принимать такое решение каждый раз (как это делается в C++). Вы должны заранее подумать о способах использования ваших типов и сделать правильный выбор семантики.

ВЛАДИМИР РЕШЕТНИКОВ

C# также поддерживает небезопасные типы указателей, которые описаны в конце данной спецификации. Они называются «небезопасными», поскольку неаккуратное обращение с ними может привести к таким нарушениям безопасности типов, которые не сможет распознать компилятор.

Типы-значения в C# подразделяются на *простые*, *перечислимые*, *структурные* и *обнуляемые*. Ссылочные типы делятся на *классы*, *интерфейсы*, *массивы* и *делегаты*. Система типов C# представлена в следующей таблице.

Категория	**Описание**	
Типы-значения	Простые типы	Целое со знаком: `sbyte`, `short`, `int`, `long`
		Целое без знака: `byte`, `ushort`, `uint`, `ulong`
		Символы Unicode: `char`
		Плавающая запятая IEEE: `float`, `double`
		Десятичная дробь с большой точностью: `decimal`
		Булевское выражение: `bool`
	Перечисления	Определяемые пользователем типы в виде `enum E {...}`
	Структуры	Определяемые пользователем типы в виде `struct S {...}`
	Обнуляемые типы	Расширения всех остальных типов-значений добавлением значения `null`
Ссылочные типы	Классы	Первичный базовый класс для всех других типов `object`
		Строки Unicode: `string`
		Определяемые пользователем типы в виде `class C {...}`
	Интерфейсы	Определяемые пользователем типы в виде `interface I {...}`
	Массивы	Одномерные и многомерные; например `int[]` и `int[,]`
	Делегаты	Определяемые пользователем типы в виде, например `delegate int D(...)`

Восемь целых типов обеспечивают поддержку 8-битных, 16-битных, 32-битных и 64-битных значений со знаком и без знака.

ДЖОН СКИТ

`Byte` теперь тип без знака, ура! Тот факт, что в Java `byte` со знаком (и не имеет эквивалента без знака), породил множество ошибок при программировании на уровне битов, которых можно было избежать.
Вполне возможно, что нам всем следует чаще использовать `uint`, чем это обычно делается: я уверен, что многие разработчики используют по умолчанию `int`, когда им нужен целый тип. Создатели каркаса, конечно, также попадают в эту категорию: зачем нужен знак для `String.Length`?

ЭРИК ЛИППЕРТ

Ответ на вопрос Джона состоит в том, что каркас[1] создавался таким образом, чтобы он мог хорошо работать с общеязыковой спецификацией CLS. CLS определяет ряд основных свойств, которыми желательно обладать CLS-совместимому языку: в подмножестве CLS целые без знака не используются.

[1] Имеется в виду .NET Framework. — *Примеч. перев.*

Два типа с плавающей запятой, float и double, используются для представления 32-битного формата с одинарной точностью и 64-битного формата с двойной точностью в соответствии со стандартом IEEE 754.

Тип decimal представляет собой 128-битный тип данных для финансовых вычислений.

ДЖОН СКИТ

Два последних абзаца можно понять так, что decimal не является типом с плавающей запятой. На самом деле он является *десятичным* типом с плавающей запятой, тогда как float и double являются *двоичными* типами с плавающей запятой.

Тип bool в C# используется для представления булевских значений — значений true или false.

Символы и строки представляются с помощью кодировки Unicode. Тип char представлен кодовой единицей UTF-16, а тип string представлен последовательностью кодовых единиц UTF-16.

Следующая таблица кратко описывает числовые типы C#.

Категория	Битов	Тип	Диапазон/Точность
Целое со знаком	8	sbyte	–128...127
	16	short	–32 728...32 767
	32	int	–2 147 483 648...2 147 483 647
	64	long	–9 223 372 036 854 775 808... 9 223 372 036854 775 807
Целое без знака	8	byte	0...255
	16	ushort	0...65 535
	32	uint	0...4 294 967 295
	64	ulong	0...18 446 744 073 709 551 615
Плавающая запятая	32	float	от 1.5×10^{-45} до 3.4×10^{38}, точность 7 разрядов
	64	double	от 5.0×10^{-324} до 7.9×10^{308}, точность 15 разрядов
Десятичный	128	decimal	от 1.0×10^{-28} до 7.9×10^{28}, точность 28 разрядов

КРИСТИАН НЕЙГЕЛ

Одна из проблем при работе с C++ на различных платформах — то, что стандарт не определяет число битов, используемых для типов short, int и long. Стандарт определяет только что short <= int <= long, что приводит к разным размерам на 16-битной, 32-битной и 64-битной платформах. В C# длина числовых типов определена явно, и неважно, какая используется платформа.

Для создания новых типов программы на C# используют *объявления типов*. В объявлении типа определяется имя типа и его элементы. Пользователи C# могут определять пять категорий типов: классы, структуры, интерфейсы, перечисления и делегаты.

Класс определяет структуру данных, содержащую данные (поля) и функциональные элементы (методы, свойства и др.). Классы поддерживают единичное наследование и полиморфизм, механизмы, посредством которых можно создавать производные классы, расширяющие и специализирующие базовые классы.

> **ЭРИК ЛИППЕРТ**
>
> Выбор для классов одиночного наследования вместо множественного устраняет одним ударом массу неудобств, возникающих в языках, использующих множественное наследование.

Структурный тип подобен классу в том, что он представляет собой структуру с элементами-данными и элементами-функциями. Однако, в отличие от классов, структуры являются типами-значениями и не требуют размещения в куче. Структуры не поддерживают наследование, определяемое пользователем, и все они неявно происходят от типа `object`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Структуры происходят от `object` не напрямую. Их прямой базовый класс есть `System.ValueType`, который, в свою очередь, является прямым наследником `object`.

Интерфейсный тип определяет контракт как именованное множество открытых (`public`) функций. Класс или структура, реализующие интерфейс, должны обеспечивать реализацию функций, являющихся элементами интерфейса. Интерфейс может наследоваться от нескольких базовых интерфейсов, и класс или структура могут реализовывать множество интерфейсов.

Делегат представляет собой ссылки на методы с определенным списком параметров и типом результата. Делегаты дают возможность использовать методы как сущности, которые могут быть присвоены переменным и передаваться как параметры. Делегаты напоминают понятие указателей функций, существующее в некоторых других языках, но, в отличие от указателей функций, делегаты являются объектно-ориентированными и типобезопасными.

Все эти типы: классы, структуры, интерфейсы и делегаты существуют и в обобщенной форме и таким образом могут быть параметризованы другими типами.

Перечисление — это отдельный тип, содержащий именованные константы. Каждый перечислимый тип имеет в основе базовый тип, который должен быть одним из восьми целых типов. Множество значений перечисления совпадает с множеством значений базового типа.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Перечисления не могут содержать в своем описании параметры-типы. Однако они могут быть параметризованы типом, если располагаются внутри обобщенного типа — класса или структуры. Более того, C# поддерживает указатели на обобщенные перечисления в небезопасном коде.

C# поддерживает одномерные и многомерные массивы любого типа. В отличие от типов, перечисленных выше, массивы не нужно объявлять до использования. Они создаются путем добавления квадратных скобок после имени типа. Например, `int[]` является одномерным массивом, состоящим из `int`, `int[,]` является двумерным массивом, состоящим из `int`, а `int[][]` будет одномерным массивом, состоящим из одномерных массивов `int`.

Обнуляемые типы также не нужно объявлять до использования. Для каждого необнуляемого значимого типа `T` существует соответствующий обнуляемый тип `T?`, который может содержать добавочное значение `null`. Например, `int?` является типом, который может содержать любое 32-битное целое число или значение `null`.

КРИСТИАН НЕЙГЕЛ

`T?` есть краткое обозначение структуры `Nullable<T>`.

ЭРИК ЛИППЕРТ

В C# 1.0 были обнуляемые ссылочные типы и необнуляемые типы-значения. В C# 2.0 мы добавили обнуляемые типы-значения. Но теперь нет необнуляемых ссылочных типов. Если мы должны будем переделывать это снова, мы, возможно, сможем добавлять обнуляемость и необнуляемость к системе типов хоть каждый день. К несчастью, необнуляемые ссылочные типы добавить трудно, так как существующая система типов для этого не приспособлена. Мы постоянно получаем запросы на создание необнуляемых ссылочных типов; это было бы замечательное свойство. Однако кодовые контракты (code contracts) далеко продвинулись по пути решения проблем, решаемых с помощью необнуляемых ссылочных типов; если вы хотите внедрить необнуление в свои программы, обратитесь к их использованию. Если этот предмет вас интересует, вы можете также поработать с Spec#, пробной версией Microsoft C#, которая поддерживает необнуляемые ссылочные типы.

C# имеет единую систему типов, так что значение любого типа может использоваться как объект. Любой тип в C# прямо или косвенно наследуется от типа `object`, и `object` является исходным базовым классом всех типов. Значения ссылочных типов рассматриваются как объекты просто путем их представления как величин типа `object`. Значения типов-значений преобразуются в объекты с помощью выполнения операций *упаковки* и *распаковки*. В следующем примере значение `int` преобразуется в `object` и обратно в `int`.

```
using System;
class Test
{
    static void Main() {
        int i = 123;
        object o = i;               // Упаковка
        int j = (int)o;             // Распаковка
    }
}
```

Когда значение типа-значения преобразуется к типу `object`, для хранения значения создается экземпляр объекта, называемый упаковкой (box), и значение копируется в эту упаковку. Наоборот, когда `object` преобразуется к типу-значению, происходит проверка, что данный объект является упаковкой допустимого типа-значения, и если она проходит успешно, значение копируется из упаковки.

Единая система типов в C# на деле означает, что типы-значения могут стать объектами «по требованию». Вследствие унификации библиотеки общего назначения, которые содержат тип `object`, могут использоваться как ссылочными типами, так и типами-значениями.

В C# есть несколько видов *переменных:* поля, элементы массивов, локальные переменные и параметры. Переменные представляют собой ячейки для хранения, и каждая переменная имеет тип, который определяет, какие значения могут в ней храниться, как это показано в нижеследующей таблице.

Тип переменной	**Возможное содержимое**
Необнуляемый тип-значение	Значение соответствующего типа
Обнуляемый тип-значение	Нулевое значение или значение соответствующего типа
`object`	Нулевая ссылка, ссылка на объект любого ссылочного типа или ссылка на упакованное значение любого значимого типа
Класс	Нулевая ссылка, ссылка на экземпляр этого класса или ссылка на экземпляр класса, производного от этого
Интерфейс	Нулевая ссылка, ссылка на экземпляр класса, который реализует данный интерфейс, или ссылка на упакованное значение типа-значения, который реализует данный интерфейс
Массив	Нулевая ссылка, ссылка на экземпляр массива этого типа или ссылка на экземпляр массива совместимого типа
Делегат	Нулевая ссылка или ссылка на экземпляр делегата

1.4. Выражения

Выражения составляются из операндов и знаков операций. Знаки операции в выражении показывают, какие операции применяются к операндам. Примеры знаков операций: `+`, `-`, `*`, `/` и `new`. Примеры операндов: константы, поля, локальные переменные и выражения.

Когда в выражении содержится много операций, порядок, в котором выполняются отдельные операции, определяется *приоритетом*. Например, выражение `x + y * z` вычисляется как `x + (y * z)`, поскольку операция `*` имеет более высокий приоритет по сравнению с операцией `+`.

Большинство операций могут быть *перегружены*. Перегрузка позволяет задать пользовательскую реализацию операции, если один или оба операнда относятся к определяемому пользователем классу или структуре.

ЭРИК ЛИППЕРТ

Порядок выполнения операций, управляемый приоритетом, — не тот порядок, в котором вычисляются операнды. Операнды вычисляются слева направо, точка. В предыдущем примере сначала будет вычисляться `x`, затем `y`, затем `z`, затем будет выполняться умножение, затем сложение. Вычисление операнда `x` произойдет раньше, чем `y`, потому что `x` находится слева от `y`; выполнение операции умножения произойдет раньше, чем сложения, потому что умножение имеет более высокий приоритет.

Следующая таблица кратко описывает операции C#. Категории операций расположены в порядке убывания приоритета. Операции одной категории имеют одинаковый приоритет.

Категория	**Выражение**	**Описание**
Первичные	`x.m`	Доступ к элементу
	`x(…)`	Вызов метода или делегата
	`x […]`	Доступ к массиву или индексатору
	`x++`	Постфиксный инкремент
	`x--`	Постфиксный декремент
	`new T(...)`	Создание объекта или делегата
	`new T(...){...}`	Создание объекта с инициализатором
	`new {...}`	Анонимный инициализатор объекта
	`new T[...]`	Создание массива
	`typeof(T)`	Получение объекта `System.Type` для `T`
	`checked(x)`	Вычисление выражений в проверяемом контексте
	`unchecked(x)`	Вычисление выражений в непроверяемом контексте
	`default(T)`	Получение значения по умолчанию типа `T`
	`delegate {...}`	Анонимная функция (анонимный метод)
Унарные	`+x`	Тождество
	`-x`	Отрицание
	`!x`	Логическое отрицание
	`~x`	Побитовое (поразрядное) отрицание
	`++x`	Префиксный инкремент
	`--x`	Префиксный декремент
	`(T)x`	Явное преобразование x к типу `T`
Мультипликативные	`x * y`	Умножение
	`x / y`	Деление
	`x % y`	Остаток
Аддитивные	`x + y`	Сложение, сцепление строк, комбинация делегатов
	`x – y`	Вычитание, удаление делегатов
Сдвиг	`x << y`	Сдвиг влево
	`x >> y`	Сдвиг вправо

Категория	**Выражение**	**Описание**
Проверка отношений и типов	`x < y`	Меньше
	`x > y`	Больше
	`x <= y`	Меньше или равно
	`x >= y`	Больше или равно
	`x is T`	Возвращает `true`, если `x` есть `T`, в противном случае возвращает `false`
	`x as T`	Возвращает `x` как переменную типа `T` или `null`, если `x` не является `T`
Равенство	`x == y`	Равно
	`x != y`	Не равно
Логическое `AND`	`x & y`	Целочисленное поразрядное `AND`, булевское логическое `AND`
Логическое `XOR`	`x ^ y`	Целочисленное поразрядное `XOR`, булевское логическое `XOR`
Логическое `OR`	`x | y`	Целочисленное поразрядное `OR`, булевское логическое `OR`
Условное `AND`	`x && y`	Вычисляется `y` только если `x` есть `true`
Условное `OR`	`x || y`	Вычисляется `y` только если `x` есть `false`
Нулевое объединение	`x ?? y`	Вычисляется `y`, если `x` есть `null`, `x` в противном случае
Условный	`x ? y : z`	Вычисляется `y` если `x` есть `true`, `z` если `x` есть `false`
Присваивание и анонимные функции	`x = y`	Присваивание
	`x op= y`	Сложное присваивание; поддерживаемые операции: `*= /= %= += -= <<= >>= &= ^= |=`
	`(T x) => y`	Анонимные функции (лямбда-выражения)

ЭРИК ЛИППЕРТ

Часто вызывает удивление, что лямбда-выражения и анонимные методы синтаксически описываются как операции. Они являются необычными операциями. Более привычно представлять себе операции как выражения с операндами, а не как объявление формальных параметров. Синтаксически, однако, лямбда или анонимный метод — такая же операция, как любая другая.

1.5. Операторы

Действия программы выражаются с помощью операторов. C# поддерживает несколько видов операторов, многие из которых определяются в терминах вложенных операторов.

Блок позволяет описывать несколько операторов в контексте, в котором разрешен только один. Блок состоит из списка операторов, записанных между разделителями { и }.

Операторы описания используются для объявления локальных переменных и констант.

Операторы-выражения используются для вычисления выражений. Выражения, которые могут быть использованы как самостоятельные операторы: вызов метода, размещение объекта с помощью операции `new`, обычные и сложные (составные) присваивания, использующие операцию `=`, а также инкремент и декремент, использующие операции `++` и `--`.

Операторы выбора используются для выбора одного из возможных операторов для выполнения на основе значения некоторого выражения. В этой группе находятся операторы `if` и `switch`.

Операторы цикла применяются для осуществления повторяющегося выполнения вложенных операторов. В этой группе операторов находятся `while`, `do`, `for` и `foreach`.

Операторы перехода используются для передачи управления. В этой группе находятся `break`, `continue`, `goto`, `throw`, `return` и `yield`.

Оператор `try...catch` используется для обнаружения исключительных ситуаций, возникших при выполнении блока, а оператор `try...finally` используется для определения завершающего кода, который выполняется всегда, независимо от того, были исключительные ситуации или их не было.

> **ЭРИК ЛИППЕРТ**
>
> Это не вся правда; конечно, блок `finally` выполняется не всегда. Код в блоке `try` может зациклиться, исключительные ситуации могут инициировать «fail fast» («мгновенный провал»), который прекратит процесс без всякого выполнения блока `finally` или кто-нибудь выдернет шнур из розетки.

Операторы `checked` и `unchecked` используются для управления контекстом контроля переполнения при целочисленных арифметических операциях и преобразованиях типа.

Оператор `lock` используется для того, чтобы получить взаимоисключающую блокировку для данного объекта, выполнить оператор, а затем снять блокировку.

Оператор `using` используется для того, чтобы получить ресурс, выполнить оператор и затем освободить ресурс.

В нижеследующей таблице дан список операторов C# и соответствующие примеры.

Оператор	**Пример**
Объявление локальной переменной	`static void Main() {` `    int a;` `    int b = 2, c = 3;` `    a = 1;` `    Console.WriteLine(a + b + c);` `}`

<table>
<tr><th>Оператор</th><th>Пример</th></tr>
<tr><td>Объявление локальной константы</td><td><pre><code>static void Main() {
     const float pi = 3.1415927f;
     const int r = 25;
     Console.WriteLine(pi * r * r);
}</code></pre></td></tr>
<tr><td>Операторы-выражения</td><td><pre><code>static void Main() {
     int i;
     i = 123;   // Оператор-выражение
     Console.WriteLine(i); // Оператор-выражение
     i++;   // Оператор-выражение
     Console.WriteLine(i); // Оператор-выражение
}</code></pre></td></tr>
<tr><td>Операторы <code>if</code></td><td><pre><code>static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("Аргументов нет");
    }
    else {
       Console.WriteLine("Один или более аргументов");
    }
}</code></pre></td></tr>
<tr><td>Оператор <code>switch</code></td><td><pre><code>static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("Аргументов нет");
            break;
        case 1:
            Console.WriteLine("Один аргумент");
            break;
        default:
            Console.WriteLine("{0} аргументов", n);
            break;
        }
    }
}</code></pre></td></tr>
<tr><td>Оператор <code>while</code></td><td><pre><code>static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}</code></pre></td></tr>
<tr><td>Оператор <code>do</code></td><td><pre><code>static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}</code></pre></td></tr>
<tr><td>Оператор <code>for</code></td><td><pre><code>static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}</code></pre></td></tr>
</table>

продолжение ⇨

<table>
<tr><th>Оператор</th><th>Пример</th></tr>
<tr><td>Оператор foreach</td><td><pre>static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}</pre></td></tr>
<tr><td>Оператор break</td><td><pre>static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}</pre></td></tr>
<tr><td>Оператор continue</td><td><pre>static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}</pre></td></tr>
<tr><td>Оператор return</td><td><pre>static int Add(int a, int b) {
return a + b;

static void Main() {
Console.WriteLine(Add(1, 2));
return;
}</pre></td></tr>
<tr><td>Оператор yield</td><td><pre>static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}
static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}</pre></td></tr>
<tr><td>Операторы throw и try</td><td><pre>static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}
static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Требуется 2 аргумента");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }</pre></td></tr>
</table>

Оператор	**Пример**
	`finally {` `Console.WriteLine("До свидания!");` `}` `}`
Операторы `checked` и `unchecked`	`static void Main() {` `    int i = int.MaxValue;` `    checked {` `        Console.WriteLine(i + 1); // Исключение` `    }` `    unchecked {` `        Console.WriteLine(i + 1); // Переполнение` `    }` `}`
Оператор `lock`	`class Account` `{` `    decimal balance;` `    public void Withdraw(decimal amount) {` `        lock (this) {` `            if (amount > balance) {` `                throw new Exception("Недостаточно средств");` `            }` `            balance -= amount;` `        }` `    }` `}`
Оператор `using`	`static void Main() {` `    using (TextWriter w = File.CreateText("test.txt")) {` `        w.WriteLine("Строка один");` `        w.WriteLine("Строка два");` `        w.WriteLine("Строка три");` `    }` `}`

1.6. Классы и объекты

Классы являются самым основным типом в C#. Класс представляет собой структуру данных, которая сочетает в себе состояние (поля) и действия (методы и другие функциональные элементы). Класс обеспечивает определение для динамически создаваемых *экземпляров* класса, также называемых *объектами*. Классы поддерживают *наследование* и *полиморфизм*, механизмы, с помощью которых *производные классы* могут расширять и специализировать *базовые классы*.

Новые классы создаются с помощью объявлений классов. Объявления классов начинаются с заголовка, который определяет атрибуты и модификаторы класса, имя класса, базовый класс (если есть) и интерфейсы, реализуемые классом. За заголовком следует тело класса, которое включает в себя список объявлений элементов класса, записанных между разделителями { и }.

Далее следует пример объявления простого класса с именем `Point`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Экземпляры классов создаются с помощью операции `new`, которая выделяет память для нового экземпляра, вызывает конструктор для инициализации экземпляра и возвращает ссылку на экземпляр. Следующие операторы создают два объекта `Point` и хранят ссылки на эти объекты в двух переменных:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

Когда объект больше не используется, память, занимаемая объектом, автоматически освобождается. Явное освобождение памяти из-под объектов не является в C# ни необходимым, ни возможным.

1.6.1. Элементы класса

Элементы класса являются либо статическими, либо элементами экземпляров. Статические элементы принадлежат классам, элементы экземпляров принадлежат объектам (экземплярам классов).

ЭРИК ЛИППЕРТ

Термин «статический» был выбран в основном из-за того, что он понятен людям, пишущим на подобных языках, хотя он не особенно точно описывает смысл понятия «разделяемый всеми элементами класса».

ДЖОН СКИТ

Я считаю, что термин «разделяемый» (в том же смысле, как в Visual Basic) тоже создает неправильное впечатление. «Разделяемый» воспринимается как нечто, требующее нескольких участников, тогда как статический элемент не относится *ни к какому* экземпляру типа. У меня есть идеальный термин для этой ситуации, но слишком длинный для того, чтобы заменить «статический» на «ассоциируемый-с-типом-а-не-с-любым-отдельным-экземпляром-типа» (дефисы не обязательны).

В следующей таблице дан обзор видов элементов классов.

Элементы	**Описание**
Константы	Постоянные значения, ассоциируемые с классом
Поля	Переменные класса
Методы	Вычисления и действия, которые могут быть выполнены классом

Элементы	Описание
Свойства	Действия, связанные с записью и чтением именованных характеристик класса
Индексаторы	Действия, связанные с индексированием экземпляров класса аналогично массивам
События	Уведомления, которые могут быть сгенерированы классом
Операции	Преобразования типов и операции, поддерживаемые классом
Конструкторы	Действия, требующиеся для инициализации экземпляров класса или всего класса в целом
Деструкторы	Действия, которые необходимо выполнить, прежде чем экземпляры класса будут окончательно удалены
Типы	Вложенные типы, объявленные в классе

1.6.2. Доступность

Каждый элемент класса имеет связанный с ним вид доступа, который определяет, из каких мест текста программы можно обратиться к данному элементу. В следующей таблице приведены пять возможных видов доступа.

Доступность	Значение
`public`	Доступ не ограничен
`protected`	Доступ ограничен этим классом и производными от него
`internal`	Доступ ограничен данной программой
`protected internal`	Доступ ограничен данной программой или классами, производными от данного класса
`private`	Доступ ограничен данным классом

КШИШТОФ ЦВАЛИНА

Следует быть осторожным с ключевым словом `public`. В C# это не то же самое, что `public` в C++! В C++ это ключевое слово означает «внутренний для единицы компиляции». В C# оно означает то же, что `extern` в C++ (то есть что каждый может его вызвать). Это огромная разница!

КРИСТИАН НЕЙГЕЛ

Я описал бы модификатор доступа `internal` как «доступ ограничен сборкой», а не как «доступ ограничен данной программой». Если этот модификатор используется внутри DLL, EXE-файлы, ссылающиеся на DLL, не имеют к ней доступа.

ЭРИК ЛИППЕРТ

`protected internal` может оказаться сомнительным и иногда неудачным выбором. Многие люди используют этот ключ некорректно, поскольку полагают, что `protected internal` означает «доступ, ограниченный производными классами в этой программе». Они думают, что это означает большую степень ограничения, тогда как на самом деле

продолжение ⇨

> это означает меньшую степень ограничения. Для того чтобы помнить об этом, полезно не забывать, что «естественное» состояние есть `private` и любой модификатор доступа *расширяет* область доступа.

Если бы гипотетическая будущая версия C# обеспечила синтаксис для «более ограничивающей комбинации protected и internal», встал бы вопрос, какая комбинация ключей будет выражать это значение. Лично я стою за «proternal» или «intected», но предполагаю, что буду разочарован.

> **КРИСТИАН НЕЙГЕЛ**
>
> В C# `protected internal` определяется как ограничение доступа данной сборкой *или* классами, производными от данного класса. CLR позволяет ограничить доступ данной сборкой *и* классами, производными от данного класса. C++/CLI позволяет использовать это свойство CLR с помощью модификаторов доступа public private (или private public — в данном случае порядок не имеет значения). На практике этот модификатор доступа используется редко.

1.6.3. Параметры-типы

Определение класса может включать ряд параметров-типов. Для этого за именем класса указывают квадратные скобки, в которых содержится список имен параметров-типов. Параметры-типы затем могут использоваться в теле класса для определений элементов класса. В следующем примере параметрами-типами класса `Pair` являются `TFirst` и `TSecond`.

```
public class Pair<TFirst, TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Класс, который объявлен с параметрами-типами, называется обобщенным (родовым, параметризованным) классом. Структура, интерфейс и делегат также могут быть обобщенными.

> **ЭРИК ЛИППЕРТ**
>
> Если вам необходимо два, три или более параметров-типов, удобно использовать обобщенные типы «кортеж», определенные в четвертой версии CLR.

Когда используется параметризованный класс, аргументы типа должны быть обеспечены для каждого параметра-типа:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;      // TFirst - целое
string s = pair.Second;  // TSecond – типа string
```

Обобщенный тип с аргументами типа называется сконструированным типом, как `Pair<int,string>` в приведенном примере.

1.6.4. Базовые классы

Объявление класса может содержать базовый класс. Для этого за именем класса и параметрами типа указывают двоеточие и имя базового класса. Пропустить спецификацию базового класса — то же, что установить наследование от типа `object`. В следующем примере базовый класс класса `Point3D` есть `Point`, а базовым классом для `Point` является `object`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
public class Point3D : Point
{
    public int z;
    public Point3D(int x, int y, int z): base(x, y)
    {
        this.z = z;
    }
}
```

Класс наследует элементы базового класса. Наследование означает, что класс неявно содержит все элементы базового класса, за исключением конструкторов экземпляров, статических конструкторов и деструкторов базового класса. В производный класс можно добавить новые элементы к тем, которые он унаследовал, но нельзя отменить определения унаследованных элементов. В предыдущем примере `Point3D` наследует от `Point` поля `x` и `y`, и каждый экземпляр `Point3D` содержит три поля `x`, `y` и `z`.

> **ДЖЕСС ЛИБЕРТИ**
>
> Нет ничего более важного для понимания C#, чем наследование и полиморфизм. Эти понятия можно считать сердцем языка и душой объектно-ориентированного программирования. Читайте этот раздел до тех пор, пока не уловите смысл, читайте разделы помощи или дополнительную литературу, но не перескакивайте через эти вопросы — без них нет C#.

Существует неявное преобразование типа класса к типу любого из его базовых классов. Таким образом, переменные типа класса могут ссылаться на экземпляры этого класса или на экземпляры любого производного класса. Например, для описания класса, приведенного выше, переменная типа `Point` может ссылаться на `Point` или на `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

1.6.5. Поля

Поле является переменной, связанной с классом или с экземпляром класса.

Поле, объявленное с модификатором `static`, определяется как статическое поле. Статическое поле имеет ровно одно место хранения. Неважно, сколько экземпляров класса создано, копия статического поля всегда только одна.

> **ЭРИК ЛИППЕРТ**
>
> Для обобщенных классов статические поля соответствуют сконструированному типу. Так что если у вас есть описание
>
> ```
> class Stack<T> {
> public readonly static Stack<T> empty = whatever; ...
> }
> ```
>
> то `Stack<int>.empty` — это другое поле, чем `Stack<string>.empty`.

Поле, объявленное без модификатора `static`, является полем экземпляра. Каждый экземпляр класса содержит отдельную копию всех полей экземпляров этого класса.

В следующем примере каждый экземпляр класса `Color` содержит отдельную копию полей экземпляра `r`, `g`, и `b`, но есть только одна копия статических полей `Black`, `White`, `Red`, `Green` и `Blue`:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte r, g, b;

    public Color(byte r, byte g, byte b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
```

Как показано в предыдущем примере, поля *только для чтения* могут быть объявлены с модификатором `readonly`. Присваивание полю `readonly` может выполняться только при описании поля или в конструкторе того же класса.

> **БРЭД АБРАМС**
>
> `readonly` защищает место хранения поля от изменений, которые могли бы быть внесены помимо конструктора класса, но не защищает хранимое там значение. Например, рассмотрим следующий тип:
>
> ```
> public class Names
> {
>    public static readonly StringBuilder FirstBorn = new StringBuilder("Joe");
> ```

```
   public static readonly StringBuilder SecondBorn = new StringBuilder("Sue");
}
```

Прямое изменение `FirstBorn` (вне конструктора) приводит к ошибке компиляции:

```
Names.FirstBorn = new StringBuilder("Biff");    // Ошибка компиляции
```

Однако можно достичь нужного результата, модифицируя экземпляр `StringBuilder`:

```
Names.FirstBorn.Remove(0,6).Append("Biff");
Console.WriteLine(Names.FirstBorn);          // Выводится "Biff"
```

По этой причине мы настоятельно рекомендуем, чтобы поля только для чтения использовались только для неизменяемых типов. Неизменяемые типы не имеют открытых механизмов установки, как `int`, `double` или `String`.

БИЛЛ ВАГНЕР

Некоторые хорошо известные паттерны проектирования используют поля только для чтения для изменяемых типов. Наиболее известные примеры — Adapter, Decorator, Fasade и Proxy. Когда создается большая структура, состоящая из меньших структур, часто приходится представлять экземпляры меньшей структуры, используя поля только для чтения. Поле только для чтения в изменяемом типе должно говорить о том, что используется один из этих структурных паттернов.

1.6.6. Методы

Метод — это элемент, выполняющий вычисления или действия, которые могут быть произведены объектом или классом. *Статические методы* доступны через класс. *Методы экземпляра* доступны через экземпляры класса.

Методы имеют список *параметров* (он может быть пустым), который содержит значения или ссылки, переданные методу, и *тип результата* — тип значения, вычисляемого и возвращаемого методом. Если метод не возвращает значения, его тип результата `void`.

Подобно типам, методы также могут иметь ряд параметров-типов, для которых аргументы типа должны определяться при вызове метода. В отличие от типов аргументы типа часто могут быть выведены из аргументов вызываемого метода и тогда нет необходимости задавать их в явном виде.

Сигнатура метода должна быть уникальна для класса, в котором объявлен метод. Сигнатура метода включает в себя имя метода, количество параметров-типов, а также число, модификаторы и типы параметров метода. Сигнатура метода не содержит тип результата.

ЭРИК ЛИППЕРТ

Неудачным следствием параметризованных типов является то, что для обобщенных типов в сконструированном типе может оказаться два метода с одинаковыми сигнатурами.

продолжение ⇨

> Например, `class C<T> { void M(T t){} void M(int t){} ...}` совершенно законен, но `C<int>` имеет два метода `M` с идентичными сигнатурами. Как мы увидим далее, эта возможность ведет к некоторым интересным особенностям при разрешении перегрузки и явной реализации интерфейса. Полезное руководящее указание: не создавайте обобщенный тип, который может породить неопределенности в конструкциях такого сорта; такие типы крайне подвержены ошибкам и могут вести себя непредсказуемо.

1.6.6.1. Параметры

Параметры используются, чтобы передавать методам значения или ссылки на переменные. Параметры метода получают свои значения от *аргументов*, которые задаются при вызове метода. Есть четыре вида параметров: параметры-значения, параметры-ссылки, выходные параметры и параметры-массивы.

Параметр-значение используется для передачи входного параметра. Параметр-значение соответствует локальной переменной, которая получает свое начальное значение от аргумента, переданного параметру. Изменения параметра-значения не влияют на аргумент, который был передан на место параметра.

> **БИЛЛ ВАГНЕР**
>
> Утверждение, что изменение параметров-значений не влияет на аргумент, может вводить в заблуждение, поскольку методы могут изменить содержимое параметра ссылочного типа. Параметр-значение не может быть изменен, но может быть изменено содержимое объекта, на который он ссылается.

Для параметров-значений можно определить значения по умолчанию, так что соответствующие аргументы можно опускать.

Параметры-ссылки используются для передачи как входных, так и выходных параметров. Аргумент, передаваемый на место параметра-ссылки, должен быть переменной, и во время выполнения метода параметр-ссылка обращается к тому месту в памяти, где хранится аргумент. Ссылочный параметр объявляется с модификатором `ref`. Следующий пример иллюстрирует использование параметров `ref`.

```
using System;
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j); // Выводится "2 1"
    }
}
```

ЭРИК ЛИППЕРТ

Этот синтаксис должен помочь прояснить недоразумение, связанное с тем, что «передачей по ссылке» называют две вещи. Ссылочные типы имеют такое название в C#, поскольку они «переданы по ссылке»; экземпляр объекта передается методу, а метод получает ссылку на экземпляр объекта. Некоторый другой код также может ссылаться на тот же самый объект.

Ссылочные параметры представляют собой несколько другую форму «передачи по ссылке». В этом случае ссылка идет на саму переменную, а не на экземпляр объекта. Если эта переменная содержит тип-значение (как это было в приведенном ранее примере), все совершенно законно. Значение не передается по ссылке, но передается переменная, которая его содержит.

Хороший способ составить представление о ссылочном параметре — помнить, что ссылочный параметр становится *псевдонимом* для переменной, переданной в качестве аргумента. В предыдущем примере `x` и `i` по существу *одна и та же* переменная.

Выходной параметр служит для передачи данных из метода в вызывающий код. Выходной параметр подобен параметру-ссылке, за исключением того, что начальное значение аргумента, указанного при вызове, не играет роли. Выходной параметр объявляется с модификатором `out`. Следующий пример иллюстрирует использование параметров `out`.

```
using System;
class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }
    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem); // Выводится "3 1"
    }
}
```

ЭРИК ЛИППЕРТ

CLR непосредственно поддерживает только параметры `ref`. Параметр `out` представляется в виде метаданных как параметр `ref`, снабженный специальным атрибутом, который показывает компилятору C#, что с этим параметром `ref` нужно обращаться как с параметром `out`. Это объясняет, почему не разрешено иметь два метода, которые отличаются только модификаторами `out` и `ref`: с точки зрения среды выполнения это идентичные методы.

Параметр-массив позволяет передавать методу различное число аргументов. Он объявляется с модификатором `params`. Только последний параметр метода может быть параметром-массивом, и он должен быть одномерным. Хорошим примером использования параметра-массива являются методы `Write` и `WriteLine` класса `System.Console`. Они описываются следующим образом:

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

Внутри метода, который использует параметр-массив, последний ведет себя в точности как обычный параметр типа массива. Однако при вызове метода с параметром-массивом можно передать на его место либо единственный аргумент типа параметра-массива, либо некоторое количество аргументов типа элемента параметра-массива. В последнем случае экземпляр массива автоматически создается и инициализируется этими аргументами. Пример:

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

эквивалентен следующему:

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

БРЭД АБРАМС

Можно найти сходство между `params` и понятием `varargs` из С-программирования. Поскольку существовала цель сделать C# очень простым для понимания, модификатор `params` не требует специального соглашения о вызовах или специальной поддержки библиотеки. Поэтому он оказался менее подверженным ошибкам, чем `varargs`. Отметим, однако, что в модели C# при каждом вызове метода неявно выполняется дополнительное выделение памяти под объект-массив. Это редко становится проблемой, но для внутренних циклов, где это может стать неэффективным, мы предлагаем определить перегруженные варианты методов для большинства случаев, а вариант с `params` использовать только в крайних случаях. Примером является семейство перегрузок `StringBuilder.AppendFormat()`:

```
public StringBuilder AppendFormat(string format, object arg0);
public StringBuilder AppendFormat(string format, object arg0, object arg1);
public StringBuilder AppendFormat(string format, object arg0, object arg1,
object arg2);
public StringBuilder AppendFormat(string format, params object[] args);
```

КРИС СЕЛЛЗ

Одно из приятных последствий того факта, что `params` на самом деле является просто удобным сокращением, является то, что я не должен писать нечто безумное вроде следующего:

```
static object[] GetArgs() { ... }
static void Main() {
       object[] args = GetArgs();
       object x = args[0];
       object y = args[1];
```

```
        object z = args[2];
        Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
```

Таким образом, я вызвал метод и задал параметры так, чтобы компилятор вновь мог создать с ними массив. Конечно, на самом деле я должен был написать вот что:

```
static object[] GetArgs() { ... }

static void Main() {
        Console.WriteLine("x={0} y={1} z={2}", GetArgs());
}
```

Однако в наши дни в .NET можно найти все меньше и меньше методов, которые возвращают массивы, и в основном программисты предпочитают использовать `IEnumerable<T>` из-за его гибкости. Это означает, что код мог бы выглядеть примерно так:

```
static IEnumerable<object> GetArgs() { ... }
static void Main() {
Console.WriteLine("x={0} y={1} z={2}", GetArgs().ToArray());
}
```

Это было бы удобным, если бы `params` «понимал» `IEnumerable` непосредственно. Может быть, в следующий раз.

1.6.6.2. Тело метода и локальные переменные

Тело метода содержит операторы, которые выполняются при вызове метода.

В теле метода можно объявлять переменные, которые определены для данного вызова метода. Такие переменные называются *локальными переменными*. Описание локальных переменных определяет имя типа, имя переменной и, возможно, ее начальное значение. В следующем примере локальная переменная `i` объявлена с начальным значением ноль, а локальная переменная `j` объявлена без начального значения.

```
using System;
class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

В C# требуется, чтобы локальная переменная до использования была *явно присвоена*. Например, если бы в данном примере объявление `i` не содержало начального значения, компилятор сообщил бы об ошибке при последующем использовании `i`, поскольку `i` в этих точках программы не была бы явно определена.

Метод может использовать оператор return, чтобы вернуть управление вызвавшему его коду. В методе, возвращающем `void`, оператор `return` не может определяться как выражение. В методе, возвращающем какое-либо значение, оператор `return` должен включать выражение, вычисляющее возвращаемое значение.

1.6.6.3. Статические методы и методы экземпляра

Методы, описанные с модификатором `static`, называются статическими методами. Статические методы не работают с отдельными экземплярами и имеют непосредственный доступ только к статическим элементам.

> **ЭРИК ЛИППЕРТ**
>
> Доступ к элементам экземпляра для статического метода вполне законен, если экземпляр доступен.

Метод, объявленный без модификатора `static`, является методом экземпляра. Метод экземпляра работает с отдельными экземплярами и может иметь доступ как к статическим элементам, так и к элементам экземпляров. К экземпляру, для которого был вызван метод экземпляра, можно явно обращаться с помощью `this`. Ссылка на `this` в статическом методе является ошибкой.

Обратимся к классу `Entity`, который содержит как статические элементы, так и элементы экземпляра.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity()
    {
    serialNo = nextSerialNo++;
    }
    public int GetSerialNo()
    {
        return serialNo;
    }
    public static int GetNextSerialNo()
    {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value)
    {
        nextSerialNo = value;
    }
}
```

Каждый экземпляр `Entity` содержит порядковый номер (и, возможно, некоторую другую информацию, которая здесь не показана). Конструктор `Entity` (который подобен методу экземпляра) инициализирует новый экземпляр со следующим доступным порядковым номером. Поскольку конструктор является элементом экземпляра, разрешен доступ к экземпляру поля `serialNo` и к статическому полю `nextSerialNo`.

Статические методы `GetNextSerialNo` и `SetNextSerialNo` могут иметь доступ к статическому полю `nextSerialNo`, но попытка прямого доступа к экземпляру поля `serialNo` приведет к ошибке.

Следующий пример демонстрирует использование класса `Entity`:

```
using System;
class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());         // Выводится "1000"
        Console.WriteLine(e2.GetSerialNo());         // Выводится "1001"
        Console.WriteLine(Entity.GetNextSerialNo()); // Выводится "1002"
    }
}
```

Заметим, что статические методы `SetNextSerialNo` и `GetNextSerialNo` вызываются через имя класса, тогда как метод экземпляра `GetSerialNo` вызывается через имя экземпляра класса.

1.6.6.4. Виртуальные, переопределенные и абстрактные методы

Метод экземпляра, описание которого включает в себя модификатор `virtual`, называется *виртуальным методом*. Если модификатор `virtual` отсутствует, такой метод называется *невиртуальным*.

Когда вызывается виртуальный метод, тип времени выполнения экземпляра, для которого вызывается метод, определяет конкретную реализацию вызванного метода. При выполнении невиртуального метода определяющим фактором является *тип экземпляра времени компиляции*.

Виртуальный метод может быть переопределен в производном классе. Если описание метода экземпляра содержит модификатор `override`, этот метод переопределяет унаследованный виртуальный метод с такой же сигнатурой. Описание виртуального метода *вводит* новый метод, описание переопределенного метода *уточняет* уже существующий унаследованный виртуальный метод, обеспечивая его новую реализацию.

> **ЭРИК ЛИППЕРТ**
>
> Тонкость здесь состоит в том, что переопределяемый виртуальный метод продолжает рассматриваться как метод класса, который ввел его, а не как метод класса, который переопределил его. Правила разрешения перегрузки в некоторых случаях отдают предпочтение элементам более производных типов перед элементами базовых типов; переопределение метода не «двигает» метод в иерархии, к которой он принадлежит.
> В самом начале этой главы мы отмечали, что при создании C# мы никогда не забывали об управлении версиями. Это одно из тех свойств, которые помогают предупредить «синдром ломающегося базового класса», возникающий из-за проблем с версиями.

Абстрактный метод — это виртуальный метод без реализации. Абстрактный метод объявляется с модификатором `abstract` и разрешен только в классе, который тоже объявлен как абстрактный. Абстрактный метод должен быть переопределен в любом неабстрактном производном классе.

В следующем примере объявляется абстрактный класс, `Expression`, который представлен как узел дерева выражений, и три производных класса, `Constant`, `VariableReference` и `Operation`, которые являются узлами дерева выражений для констант, ссылочных переменных и арифметических операций. (Это похоже на типы дерева выражений, введенные в разделе 4.6, но не одно и то же.)

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}
public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }
    public override double Evaluate(Hashtable vars) {
        return value;
    }
}
public class VariableReference: Expression
{
    string name;
    public VariableReference(string name) {
        this.name = name;
    }
    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Неизвестная переменная: " + name);
        }
        return Convert.ToDouble(value);
    }
}
public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }
```

```
    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Неизвестная операция");
    }
}
```

Предыдущие четыре класса могут использоваться для моделирования арифметических выражений. Например, если пользоваться экземплярами этих классов, выражение `x + 3` может быть представлено следующим образом:

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

Метод экземпляра `Expression` вызывается для того, чтобы вычислить данное выражение и получить значение `double`. Метод использует в качестве аргумента `Hashtable`, который содержит имена переменных (ключи входов) и значения (значения входов). Метод `Evaluate` является абстрактным методом; это означает, что неабстрактные производные классы должны переопределять его, чтобы обеспечить конкретную реализацию.

При выполнении метода `Constant` просто возвращается хранящаяся константа. Реализация метода `VariableReference` ищет имя переменной в хеш-таблице и возвращает значение результата. При вызове метода `Operation` сначала вычисляются левый и правый операнды, а затем выполняется заданная арифметическая операция.

Приведенная далее программа использует классы `Expression` для вычисления выражения `x * (y + 2)` для различных значений `x` и `y`.

```
using System;
using System.Collections;
class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars)); // Выводится "21"
```

продолжение ☞

```
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars)); // Выводится "16.5"
    }
}
```

КРИС СЕЛЛЗ

Виртуальные функции являются одной из главных особенностей объектно-ориентированного программирования, которая отличает его от других видов программирования. Например, предположим, что вы делаете нечто вроде:

```
double GetHourlyRate(Person p) {
    if( p is Student ) { return 1.0; }
    else if( p is Employee ) { return 10.0; }
    return 0.0;
}
```

Вместо этого можно почти всегда использовать виртуальный метод:

```
class Person {
        public virtual double GetHourlyRate() {
            return 0.0;
        }
}
class Student {
        public override double GetHourlyRate() {
            return 1.0;
        }
}
class Employee {
        public override double GetHourlyRate() {
            return 10.0;
        }
}
```

1.6.6.5. Перегрузка методов

Перегрузка методов позволяет нескольким методам одного и того же класса иметь одно и то же имя, если они имеют уникальные сигнатуры. При компиляции кода вызова перегруженного метода компилятор использует разрешение перегрузки, чтобы определить тот метод, который должен быть выполнен. Разрешение перегрузки находит метод, который лучше всего соответствует аргументам, или сообщает об ошибке, если единственный лучший метод не был найден. Следующий пример показывает разрешение перегрузки в действии. Комментарии к каждой реализации метода `Main` показывают, какой именно метод фактически вызывается.

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }
        static void F(object x) {
            Console.WriteLine("F(object)");
```

```
        }
        static void F(int x) {
            Console.WriteLine("F(int)");
        }
        static void F(double x) {
            Console.WriteLine("F(double)");
        }
        static void F<T>(T x) {
            Console.WriteLine("F<T>(T)");
        }
        static void F(double x, double y) {
            Console.WriteLine("F(double, double)");
        }
        static void Main() {
            F();          // Вызывается F()
            F(1);         // Вызывается F(int)
            F(1.0);       // Вызывается F(double)
            F("abc");     // Вызывается F(object)
            F((double)1); // Вызывается F(double)
            F((object)1); // Вызывается F(object)
            F<int>(1);    // Вызывается F<T>(T)
            F(1, 1);      // Вызывается F(double, double)
            }
}
```

Как видно из примера, конкретный метод всегда можно выбрать явным преобразованием типа аргументов к типам параметров и (или) явным заданием типа аргументов.

БРЭД АБРАМС

Возможно неправильное использование перегрузки методов. Вообще говоря, лучше использовать перегрузку только если все методы семантически одинаковы. Многие разработчики при использовании перегруженных методов воспринимают их совокупность как один метод, имеющий множество различных аргументов. Фактически изменение типа локальной переменной, параметра или свойства может привести к вызову другого варианта перегруженного метода. Многие разработчики не замечают побочных эффектов своего решения использовать перегрузку. Для пользователей, однако, удивительно, когда методы с одним и тем же именем делают разные вещи. Например, на заре создания .NET Framework (еще до выпуска первой версии) мы использовали следующие варианты перегруженных методов класса `string`:

```
public class String {
    public int IndexOf (string value);
        // Возвращает позицию value в данном экземпляре
    public int IndexOf (char value);
        // Возвращает позицию value в данном экземпляре
    public int IndexOf (char [] value);
        // Возвращает позицию самого левого из всех символов value
        // в текущем экземпляре
}
```

Последняя перегрузка приводит к проблемам, так как имеет дело с различными вещами.

продолжение ⇗

Например,

```
"Joshua, Hannah, Joseph".IndexOf("Hannah");// Возвращает 7,
```

Но

```
"Joshua, Hannah, Joseph".IndexOf(new char [] {'H','a','n','n','a,'h;");
// Возвращает 3
```

В данном случае методу, выполняющему что-либо иное, лучше дать другое имя:

```
public class String {
    public int IndexOf (string value);
        // Возвращает позицию value в данном экземпляре
    public int IndexOf (char value);
        // Возвращает позицию value в данном экземпляре
    public int IndexOfAny(char [] value);
        // Возвращает позицию самого левого из всех символов value
        // в текущем экземпляре
}
```

БИЛЛ ВАГНЕР

Перегруженные методы и наследование не очень хорошо сочетаются. Поскольку правила разрешения перегрузки в некоторых случаях отдают предпочтение методу, объявленному в производном классе самого нижнего уровня, это иногда может означать, что метод, описанный в производном классе, будет выбран вместо метода базового класса, который подошел бы лучше. По этой причине я рекомендую не перегружать элементы, описанные в базовом классе.

1.6.7. Другие функциональные элементы класса

Элементы, которые содержат исполняемый код, имеют общее название — *функциональные элементы* класса. В предыдущем разделе были описаны методы, они представляют собой основную разновидность функциональных элементов. В данном разделе описаны другие виды функциональных элементов C#: конструкторы, свойства, индексаторы, события, операции и деструкторы.

В следующей таблице приведен обобщенный класс `List<T>`, который реализует наращиваемый список объектов. Класс содержит несколько примеров наиболее распространенных видов функциональных элементов.

`public class List<T>` `{`	
`const int defaultCapacity = 4;`	Константа
`T[] items;` `int count;`	Поля
`public List(int capacity = defaultCapacity) {` `items = new T[capacity];` `}`	Конструкторы
`public int Count {` `get { return count; }` `}`	Свойства

```
     public int Capacity {
      get {
         return items.Length;
         }
      set {
         if (value < count) value = count;
         if (value != items.Length) {
            T[] newItems = new T[value];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
         }
      }
   }
```

Индексатор

```
   public T this[int index] {
      get {
         return items[index];
      }
      set {
         items[index] = value;
         OnChanged();
      }
   }
```

Методы

```
   public void Add(T item) {
      if (count == Capacity) Capacity = count * 2;
      items[count] = item;
      count++;
      OnChanged();
   }
   protected virtual void OnChanged() {
      if (Changed != null) Changed(this, EventArgs.Empty);
   }
   public override bool Equals(object other) {
      return Equals(this, other as List<T>);
   }
   static bool Equals(List<T> a, List<T> b) {
      if (a == null) return b == null;
      if (b == null || a.count != b.count) return false;
      for (int i = 0; i < a.count; i++) {
         if (!object.Equals(a.items[i], b.items[i])) {
         return false;
         }
      }
      return true;
   }
```

Событие

```
   public event EventHandler Changed;
```

Операции

```
   public static bool operator ==(List<T> a, List<T> b) {
      return Equals(a, b);
   }
   public static bool operator !=(List<T> a, List<T> b) {
      return !Equals(a, b);
   }
```

```
}
```

1.6.7.1. Конструкторы

C# поддерживает как статические конструкторы, так и конструкторы экземпляров. Конструктор экземпляра — это элемент класса, который выполняет действия, требующиеся для инициализации экземпляра класса. Статический конструктор выполняет действия, требующиеся для инициализации самого класса, когда тот впервые загружается.

Конструктор описывается подобно методу, он не возвращает значение и его имя совпадает с именем класса. Если объявление конструктора содержит модификатор `static`, оно описывает статический конструктор. В противном случае оно объявляет конструктор экземпляра.

Конструкторы экземпляров могут быть перегружены. Например, в классе `List<T>` объявлено два конструктора экземпляра, один без параметров и один с параметром `int`. Конструкторы экземпляра вызываются при помощи операции `new`. Следующие операторы создают два экземпляра `List<string>`; каждый из них использует соответствующий конструктор класса `List`.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

В отличие от других элементов, конструкторы не наследуются, и класс не имеет иных конструкторов экземпляра, кроме тех, которые были описаны в этом классе. Если для класса не был объявлен ни один конструктор, автоматически создается один пустой конструктор без параметров.

БРЭД АБРАМС

Конструкторы должны быть ленивыми! Наилучший способ минимизировать работу конструкторов — просто собирать аргументы для дальнейшего использования. Например, вы можете запомнить имя файла и путь к базе данных, но не открывать эти внешние ресурсы без абсолютной необходимости. Такая практика дает уверенность, что небольшие ресурсы будут размещены на наименьшее возможное количество времени.

Я недавно сам бился над этой проблемой с классом `DataContext` в Linq to Entities. Он открывает базу данных, имя которой ему передается, вместо того чтобы делать это только по необходимости. Проводя тесты, я, передавая тестовые данные, не имел намерения открывать базу данных. Такая излишняя активность приводит к потерям производительности, а также делает сценарий более сложным.

1.6.7.2. Свойства

Свойства представляют собой естественное развитие понятия полей. Оба эти элемента класса являются именованными сущностями определенного типа, и синтаксис доступа к полям и свойствам одинаков. Однако в отличие от полей, свойства не задают областей памяти для хранения данных, а содержат коды доступа, представляющие собой операторы, выполняющиеся при записи или считывании значения свойства.

ДЖЕСС ЛИБЕРТИ

Для создателя класса свойство подобно методу, позволяющему разработчику добавлять некоторое поведение до установки или получения значения, связанного со свойством. Напротив, для клиента класса свойство проявляет себя как поле, обеспечивающее прямой, свободный доступ через операцию присваивания.

ЭРИК ЛИППЕРТ

Стандартный «хороший стиль» — всегда представлять данные, подобные полям, в виде свойств с механизмами установки, а не в виде полей. В этом случае, если вы хотите добавить функциональности механизмам получения и установки (имеется в виду регистрация, связывание данных, проверка безопасности), можно легко сделать это, не вводя в недоумение пользователей, которые полагаются на то, что это поле должно быть там.

Хотя в некотором смысле подобная практика грешит против другого хорошего совета («избегайте необоснованных обобщений»), новые «автоматически реализуемые свойства» делают использование свойств в качестве части открытого интерфейса или типа более легким и естественным, чем использование полей.

КРИС СЕЛЛЗ

Эрик затронул такую интересную тему, что я хочу привести пример. Никогда не создавайте поле `public`:

```
class Cow
{
    public int Milk; // ПЛОХО!
}
```

Если вам не требуется ничего, кроме хранения в памяти, позвольте компилятору реализовать свойство:

```
class Cow
{
    public int Milk { get; set; } // ПРАВИЛЬНО!
}
```

В этом случае клиент привязывается к кодам доступа, и в дальнейшем вы сможете переписать реализацию, выполненную компилятором, с целью сделать что-нибудь этакое:

```
class Cow {
    bool gotMilk = false;
    int milk;
    public int Milk {
    get {
        if( !gotMilk ) {
            milk = ApplyMilkingMachine();
            gotMilk = true; }
       return milk;
    }
```

продолжение ⇗

```
    set {
        ApplyReverseMilkingMachine(value); // Корове это может не понравиться
        milk = value;
    }
}
...
}
```

А еще мне очень нравится следующая идиома для случаев, когда вы знаете, что в вашей программе будет использоваться вычисленное значение:

```
class Cow {
    public Cow() {
        Milk = ApplyMilkingMachine();
    }

    public int Milk { get; private set; }
    ...
}
```

В данном случае мы заранее вычислили свойство, что является напрасной тратой энергии, если мы не знаем, будем ли мы его использовать. Если бы мы знали, то обезопасили бы себя с помощью некоторого усложнения кода: удаления флага, некоторого разветвления логики и управления хранением.

БИЛЛ ВАГНЕР

Доступ к свойствам для пользователя выглядит так же, как доступ к полям, и пользователи, естественно, ожидают, что доступ будет аналогичен полям в любом случае, включая преобразования. Если механизм получения должен выполнить значительную работу (например, прочитать файл или выполнить запрос к базе данных), его нужно представлять как метод, а не как свойство. Вызывающая сторона ожидает, что метод может работать дольше.

По той же самой причине повторяющиеся вызовы кодов доступа свойства (без промежуточного кода) должны возвращать одно и то же значение. `DateTime.Now` — один из немногих примеров в каркасе, который не следует данному совету.

Свойство объявляется подобно полю, за исключением того, что описание заканчивается кодами получения (`get`) и установки (`set`), записанными между разделителями `{` и `}` вместо точки с запятой. Свойство, которое имеет и код установки, и код получения, является *свойством, доступным для чтения и записи*, свойство, для которого задан только код получения, является *свойством только для чтения*, а свойство, которое имеет только код установки, именуется *свойством только для записи*.

Код получения (`get`) соответствует методу без параметров, который возвращает значение типа свойства. За исключением использования в левой части операции присваивания, когда свойство обрабатывается как выражение, для вычисления значения свойства выполняется код `get`.

Код установки (`set`) соответствует методу, не возвращающему значение, с единственным параметром, называемым `value`. Когда свойство используется в операциях присваивания, `++` или `--`, выполняется код установки `set` с аргументом, задающим новое значение свойства.

В классе `List<T>` описано два свойства, `Count` и `Capacity`, которые являются соответственно свойством только для чтения и свойством для чтения и записи. Следующий пример демонстрирует использование этих свойств.

```
List<string> names = new List<string>();
names.Capacity = 100;              // Вызывается код установки set
int i = names.Count;               // Вызывается код получения get
int j = names.Capacity;            // Вызывается код получения get
```

Подобно полям и методам, в C# поддерживаются свойства экземпляра и статические свойства. Статические свойства объявляются с модификатором `static`, свойства экземпляра объявляются без него.

Коды доступа могут быть виртуальными. Когда описание свойства содержит модификаторы `virtual`, `abstract` или `override`, они относятся и к кодам доступа.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Если виртуальное свойство имеет код доступа, описанный как `private`, этот код доступа выполняется в CLR как невиртуальный метод и не может быть переопределен в производном классе.

1.6.7.3. Индексаторы

Индексатор дает возможность объектам быть индексированными таким же образом, как массив. Индексатор описывается подобно свойству, за исключением того, что имя элемента представляет собой `this`, за которым следует список параметров в квадратных скобках [и]. В кодах доступа индексатора можно указывать параметры. Подобно свойствам, индексаторы могут быть доступны для *чтения и записи*, *только для чтения* и *только для записи*, и коды доступа индексатора могут быть виртуальными.

В классе `List` объявлен единственный индексатор, доступный для чтения и записи, с параметром `int`. Последний дает возможность индексировать экземпляры класса `List` значением `int`. Например:

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Индексаторы могут быть перегружены. Это означает, что класс может объявлять множество индексаторов — если число или тип их параметров отличаются друг от друга.

1.6.7.4. События

События являются элементами класса, которые дают возможность классу или объекту обеспечивать уведомления. События объявляются подобно полям, за исключением того, что описание содержит ключевое слово `event` и тип должен быть типом делегата.

> **ДЖЕСС ЛИБЕРТИ**
>
> На самом деле `event` — это просто ключевое слово, которое сигнализирует C#, что нужно ограничить способ использования делегата, таким образом препятствуя клиенту инициировать события непосредственно или получить контроль над событием путем присваивания обработчика, а не его добавления. Короче говоря, ключевое слово `event` заставляет делегаты вести себя так, как должны вести себя события.

КРИС СЕЛЛЗ

Без ключевого слова `event` можно сделать следующее:

```
delegate void WorkCompleted();
class Worker {
    public WorkCompleted Completed; // Поле-делегат (не событие)
    ...
}
class Boss {
    public void WorkCompleted() { ... }
}
class Program {
    static void Main() {
        Worker peter = new Worker();
        Boss boss = new Boss();
    peter.Completed += boss.WorkCompleted;     // То, чего вы хотите
        peter.Completed = boss.WorkCompleted;  // То, что разрешит компилятор
    ...
    }
}
```

К сожалению, без ключевого слова `event` поле `Completed` является открытым полем типа делегата, которое доступно любому, кто захочет — и компилятор это допускает. Добавление ключевого слова `event` сводит операции к `+=` и `-=` следующим образом:

```
class Worker {
    public event WorkCompleted Completed;
    ...
}
...
        peter.Completed += boss.WorkCompleted; // Ошибок компиляции нет
        peter.Completed = boss.WorkCompleted;  // Ошибка компиляции
```

Использование ключевого слова `event` — это единственный правильный вариант для `public` полей, поскольку компилятор ограничит использование безопасными операциями. При этом вы можете, если хотите, управлять реализацией операций `+=` и `-+` для события.

Внутри класса, в котором описано событие, доступ к событию подобен доступу к полям типа делегата (если событие не является абстрактным и не определяет коды доступа). Поле хранит ссылку на делегат, который представляет обработчики события, добавленные к событию. Если обработчики событий отсутствуют, поле имеет значение `null`.

В классе `List<T>` объявляется единственное событие `Changed`, которое показывает, что к списку добавлен новый элемент. Событие `Changed` генерируется виртуальным методом `OnChanged`, который сначала проверяет, не `null` ли значение события (что означает, что обработчики отсутствуют). По смыслу инициирование события в точности эквивалентно вызову делегата, представленного событием, так что специальных языковых конструкций для инициирования события нет.

Клиенты реагируют на события с помощью обработчиков событий. Обработчики событий присоединяются операцией `+=` и удаляются операцией `-=`. В следующем примере обработчик присоединяется к событию `Changed` класса `List<string>`.

```
using System;
class Test
{
    static int changeCount;
    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }
    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount); // Выводится "3"
    }
}
```

Для сложных случаев, когда желательно обеспечить контроль над памятью, занимаемой событием, в объявлении события могут явным образом присутствовать коды доступа `add` и `remove`, которые в чем-то подобны коду доступа `set` для свойств.

> **КРИС СЕЛЛЗ**
>
> Со времени появления C# 2.0 явное создание экземпляра делегата для создания обертки для метода больше не нужно. Как следствие этого, код
>
> `names.Changed += new EventHandler(ListChanged);`
>
> может быть записан кратко:
>
> `names.Changed += ListChanged;`
>
> Эту краткую форму не только быстрее набирать, но и проще читать.

1.6.7.5. Операции

Операции — это элементы класса, которые определяют смысл применения определенного знака операции к экземпляру класса. Можно определить три вида опе-

раций: унарные операции, бинарные операции и операции преобразования. Все операции должны быть объявлены как `public` и `static`.

В классе `List <T>` описаны две операции: операция `==` и операция `!=`, и таким образом придается новое значение выражениям, которые применяют эти операции к экземплярам класса `List`. А именно, операции определяют равенство двух экземпляров `List<T>` путем сравнения содержащихся в нем объектов, выполняя метод `Equals`. В следующем примере используется операция `==` для сравнения двух экземпляров `List<int>`.

```
using System;
class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b); // Выводится "True"
        b.Add(3);
        Console.WriteLine(a == b); // Выводится "False"
    }
}
```

Первая операция `Console.WriteLine` выводит `True`, поскольку два списка содержат одинаковое число объектов с одинаковыми значениями, расположенными в том же самом порядке. Если бы `List<T>` не определил операцию `==`, первый `Console.WriteLine` вывел бы `False`, поскольку `a` и `b` ссылаются на разные экземпляры класса `List<int>`.

1.6.7.6. Деструкторы

Деструкторы — это элементы класса, которые выполняют действия, требующиеся для уничтожения экземпляра класса. Деструкторы не могут иметь параметров, модификаторов доступа, и они не могут быть вызваны явно. Деструктор для экземпляра вызывается автоматически во время сборки мусора.

Сборщик мусора обладает широкими полномочиями для принятия решения о том, когда собирать объекты и запускать деструктор. Конкретно время вызова деструктора не определено, и деструкторы могут быть выполнены в любом потоке. По этим и другим причинам в классах следует реализовывать деструкторы, только если нет других подходящих решений.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Деструкторы иногда называют «финализаторами» («finalizers»). Это название еще появляется в сборщике мусора API — например, `GC.WaitForPendingFinalizers`.

Оператор `using` обеспечивает лучший подход к уничтожению объектов.

1.7. Структуры

Подобно классам, структуры представляют собой структуры данных, которые содержат элементы-данные и функциональные элементы, но в отличие от классов структуры являются типом-значением и не требуют размещения в куче. Переменные типа `struct` непосредственно хранят данные, тогда как переменные типа класса хранят ссылку на динамически размещаемый объект. Структуры не поддерживают определяемое пользователем наследование, и все типы структур неявно наследуются от типа `object`.

> **ЭРИК ЛИППЕРТ**
>
> Тот факт, что структуры *не требуют* размещения в куче, не означает, что они *никогда* там не размещаются. Более подробно см. аннотации раздела 1.3.

Структуры очень полезны для небольших структур данных, которые имеют значимую семантику. Комплексные числа, точки в системе координат или пары слово–значение в словаре — все это хорошие примеры структур. Использование `struct` для маленьких структур данных значительно отличается от использования классов по количеству операций выделения памяти и производительности приложения. Например, следующая программа создает и инициализирует массив из 100 точек. Если `Point` реализован как класс, создается 101 отдельный объект — один для массива и по одному для каждого из 100 элементов.

```
class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
class Test
{
    static void Main()
    {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

Альтернатива — сделать `Point` структурой:

```
struct Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Теперь создается только один объект для массива, и экземпляры `Point` хранятся линейно расположенными в массиве.

> **ЭРИК ЛИППЕРТ**
>
> Мысль, которую нужно отсюда вынести: при работе с отдельными приложениями с большим количеством данных использование структур приносит большую пользу, так как в противном случае приложения могут подвисать при выполнении размещения в куче. Это вовсе не мысль «нужно всегда использовать структуры, поскольку они сделают так, что ваша программа будет работать быстрее».
>
> Выигрыш при выделении памяти определяется выбором: в некоторых случаях структуры требуют меньше времени для размещения и освобождения ресурса, но поскольку каждое присваивание структуры есть копия значения, копирование структуры может занять большее время, чем копирование ссылки.
>
> Всегда помните, что не имеет смысла оптимизировать что-либо, кроме того, что работает *медленнее всего*. Если ваша программа не подвисает на выделении памяти в куче, то размышлять, что лучше использовать с точки зрения выделения памяти — структуры или классы, — это неэффективный способ расходовать свое время. Найдите вещь, которая работает медленнее всего, и оптимизируйте ее.

Конструкторы структуры вызываются операцией `new`, но это не означает, что будет выделена память. Вместо динамически размещаемого объекта и возвращения ссылки на него конструктор структуры просто возвращает значение структуры (обычно временно размещенное в стеке), а затем это значение копируется по мере необходимости.

Для классов возможна ситуация, когда две переменные ссылаются на один и тот же объект и, таким образом, возможно, что операции над одной переменной действуют на объект, на который ссылается другая переменная. В случае структур каждая переменная имеет свою собственную копию данных, и невозможно, чтобы операция над одной переменной повлияла на другую. Например, вывод, выполняемый следующим фрагментом кода, зависит от того, является `Point` классом или структурой.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

Если `Point` является классом, выведется 20, потому что `a` и `b` ссылаются на один и тот же объект. Если же `Point` — структура, выведется 10, поскольку присваивание значения `a` значению `b` создает копию значения, и на эту копию не действует последующее присваивание `a.x`.

Предыдущий пример обнаруживает два ограничения структур. Во-первых, копирование целой структуры обычно менее эффективно, чем копирование ссылки на объект, так что присваивание и передача значения параметра могут обходиться для структуры дороже, чем для ссылочных типов. Во-вторых, кроме параметров `ref` и `out`, невозможно создать ссылки на структуры, что исключает их использование в некоторых ситуациях.

БИЛЛ ВАГНЕР

Перечитайте последние два абзаца. Они описывают наиболее важные конструктивные различия между структурами и классами. Если вы не хотите использовать во всех случаях семантику значений, вы должны использовать класс. Классы в некоторых ситуациях могут реализовывать семантику значений (хорошим примером является `string`), но по умолчанию они подчиняются семантике ссылок. Это различие важнее для конструирования, чем размер или различия между размещением в куче и в стеке.

1.8. Массивы

Массив является структурой данных, которая содержит некоторое число переменных, доступных с помощью вычисляемых индексов. Переменные, содержащиеся в массиве, называемые *элементами* массива, относятся к одному типу, который называется *типом элементов* массива.

Массивы являются ссылочными типами, и объявление переменной массива просто выделяет память для ссылки на экземпляр массива. Действующие экземпляры массива создаются динамически в среде выполнения при помощи операции `new`. Операция `new` определяет длину нового экземпляра массива, которая устанавливается на все время жизни экземпляра. Индексы элементов массива изменяются от нуля до `Length – 1`. Операция `new` автоматически инициализирует элементы массива их значениями по умолчанию, которое, например, есть ноль для всех численных типов и `null` для всех ссылочных типов.

ЭРИК ЛИППЕРТ

Недоразумение, возникающее от того, что некоторые языки индексируют массивы, начиная с 1, а некоторые другие — начиная с 0, сбивает с толку многие поколения начинающих программистов. Мысль, что индекс массива начинается с нуля, исходит от недопонимания тонкостей синтаксиса массивов в языках C.

В языке C, когда используется `myArray[x]`, это означает «начать с начала массива и обратиться к элементу, смещенному на `x` шагов. Таким образом, `myArray[1]` ссылается на *второй* элемент, потому что это то, что вы получили, когда начали с первого элемента и *сделали* один шаг.

На самом деле эти ссылки следовало бы называть *сдвигами*, а не *индексами*. Но поскольку поколения программистов во всем мире считают, что массивы «индексируются», начиная с нуля, мы придерживаемся этой терминологии.

В следующем примере создается массив элементов типа `int`, он инициализируется и его содержимое выводится на печать.

```
using System;
class Test
{
   static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
```

продолжение ☞

```
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

В этом примере создается *одномерный массив* и над ним выполняются действия. C# также поддерживает многомерные массивы. Количество измерений массива, также известное как размерность массива, есть один плюс число запятых, записанных между квадратными скобками типа массива. В следующем примере создаются одномерный, двумерный и трехмерный массивы.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

Массив `a1` содержит 10 элементов, массив `a2` содержит 50 (10 × 5) элементов, а массив `a3` содержит 100 (10 × 5 × 2) элементов.

БИЛЛ ВАГНЕР

Правило FxCop[1] предостерегает против многомерных массивов; это в основном касается разреженных многомерных массивов. Если вы знаете, что в самом деле все элементы в массиве заполнены, многомерные массивы вполне хороши.

Тип элемента массива может быть любым, в том числе и типом массива. Массив с элементами типа массива иногда называют ступенчатым массивом, поскольку длина элементов массива может быть разной. В следующем примере создается массив, состоящий из массивов `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

В первой строке создается массив из трех элементов, каждый из которых имеет тип `int[]` и начальное значение `null`. Следующие строки инициализируют три элемента ссылками на отдельные экземпляры массивов различной длины.

Операция `new` разрешает определение начальных значений элементов массива с помощью инициализатора массива, который представляет собой список выражений, записанных между ограничителями `{` и `}`. В следующем примере создается и инициализируется `int[]` с тремя элементами.

```
int[] a = new int[] {1, 2, 3};
```

[1] FxCop — бесплатный инструмент для статического анализа кода от Microsoft, проверяющий сборки .NET на соответствие рекомендациям по проектированию библиотек .NET Framework. — *Примеч. перев.*

Заметим, что длина массива выводится из числа выражений между фигурными скобками. Можно еще больше сократить описания локальных переменных и полей, не задавая повторно тип массива:

```
int[] a = {1, 2, 3};
```

Оба предыдущих примера эквивалентны следующему:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

ЭРИК ЛИППЕРТ

В некоторых местах спецификации указано, что локальная инициализация эквивалентна тому, чтобы «присвоить что-либо временной переменной, сделать что-нибудь с временной переменной, описать локальную переменную и присвоить временную переменную локальной». Вас может удивить такой очевидно непрямой путь, и притом без всякой необходимости. Почему бы просто не сказать, что инициализация эквивалентна следующему:

```
int[] a = new int[3];
a[0] = 1; a[1] = 2; a[2] =3;
```

На самом деле это необходимо для анализа процесса присваивания. Прежде чем использовать локальные переменные, мы должны быть уверены, что всем им присвоены значения. В частности, мы должны считать такое выражение, как `object[] arr = {arr};`, недопустимым, поскольку `arr` используется раньше, чем ему явным образом присвоено значение. Если это эквивалентно

```
object[] arr = new object[1];
arr[0] = arr;
```

тогда оно было бы законно. Но если говорить, что это выражение эквивалентно

```
object[] temp = new object[1];
temp[0] = arr;
object[] arr = temp;
```

тогда становится ясно, что `arr` используется до инициализации.

1.9. Интерфейсы

Интерфейсы определяют контракт, который может быть реализован классом или структурой. Интерфейс может содержать методы, свойства, события и индексаторы. Интерфейс не обеспечивает реализацию элементов, которые он определяет — он просто определяет элементы, которые должны быть описаны в классах или структурах, которые реализуют интерфейс.

Интерфейсы могут использовать множественное наследование. В следующем примере интерфейс `IComboBox` наследует и от `ITextBox`, и от `IListBox`.

```
interface IControl
{
    void Paint();
}
interface ITextBox : IControl
{
    void SetText(string text);
}
interface IListBox : IControl
{
    void SetItems(string[] items);
}
interface IComboBox : ITextBox, IListBox { }
```

Классы и структуры могут реализовывать множество интерфейсов. В следующем примере класс `EditBox` реализует оба интерфейса — `IControl` и `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}
public class EditBox : IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

КШИШТОФ ЦВАЛИНА

Возможно, мое утверждение несколько спорно, но я думаю, что отсутствие поддержки множественного наследования в нашей системе типов — это главная причина сложности .NET Framework. Когда мы конструировали систему типов, мы твердо решили не добавлять поддержку множественного наследования, чтобы обеспечить простоту. В ретроспективе видно, что это решение имело прямо противоположные последствия. Отсутствие множественного наследования вынудило нас добавить понятие интерфейса, которое, в свою очередь, ответственно за проблемы с развитием каркаса, углублением иерархий наследования и многими другими.

Когда класс или структура реализуют определенный интерфейс, экземпляры этого класса или структуры могут быть явно преобразованы к типу интерфейса. Например:

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Если статический тип экземпляра для реализации определенного интерфейса не известен, можно использовать динамическое приведение типов. Например, следующие операторы используют динамическое приведение типов для получения реализаций интерфейса `IControl` и `IDataBound`. Поскольку фактический тип объекта `EditBox`, приведения осуществляются успешно.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBou)obj;
```

Для приведенного выше класса `EditBox` метод `Paint` из интерфейса `IControl` и метод `Bind` из интерфейса `IDataBound` реализованы при помощи открытых (`public`) элементов. C# также поддерживает *явную реализацию элементов интерфейса,* при помощи которых класс или структура могут избежать описания элементов как `public`. Явная реализация элементов интерфейса записывается с полным именем элемента интерфейса. Например, класс `EditBox` может реализовать методы `IControl.Paint` и `IDataBound.Bind`, используя явную реализацию элемента интерфейса следующим образом:

```
public class EditBox : IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Явные элементы интерфейса могут быть доступны только через интерфейсный тип. Например, реализация `IControl.Paint` в классе `EditBox` из предыдущего примера может быть вызвана только если сначала привести ссылку на `EditBox` к интерфейсному типу `IControl`

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Ошибка: такого метода нет
IControl control = editBox;
control.Paint();                        // Правильно
```

> **ВЛАДИМИР РЕШЕТНИКОВ**
> На самом деле явно реализованные элементы интерфейса могут также быть доступны через параметр-тип, приведенный к типу интерфейса.

1.10. Перечисления

Перечисление есть отдельный тип-значение, содержащий совокупность именованных констант. В следующем примере описывается и используется перечисление, названное `Color`, с тремя константами — `Red`, `Green` и `Blue`.

```
using System;
enum Color
{
    Red,
    Green,
    Blue
}
class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
```

продолжение ☞

```
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Неизвестный цвет");
                break;
        }
}
static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Каждое перечисление имеет соответствующий целый тип, называемый *базовым типом* перечисления. Перечисление, в котором явным образом не задан базовый тип, имеет базовый тип `int`. Формат хранения и диапазон возможных значений перечисления определяются его базовым типом. Совокупность значений, которые может принимать перечисление, не ограничивается только его элементами. В частности, любое значение базового типа для данного перечисления может быть приведено к перечислению и является отдельным правильным значением этого перечисления.

В следующем примере описано перечисление, именуемое `Alignment`, с базовым типом `sbyte`.

```
enum Alignment : sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Хотя этот синтаксис похож на спецификацию родительского типа, он имеет другое значение. Родительский тип для `Alignment` не `sbyte`, а `System.Enum`, и здесь нет неявного преобразования `Alignment` к `sbyte`.

Как показано в предыдущем примере, описание перечисления может содержать константное выражение, которое определяет значение элемента. Значение константы для каждого элемента перечисления должно находиться внутри диапазона значений базового типа перечисления. Когда описание элемента перечисления не определяет значение явно, элемент получает значение ноль (если он является первым элементом перечисления) или значение, равное значению текстуально предшествующего элемента перечисления, плюс единица.

Значения перечисления могут быть с помощью приведения типов приведены к целым значениям, и наоборот. Например:

```
int i = (int)Color.Blue;                // int i = 2;
Color c = (Color)2;                     // Color c = Color.Blue;
```

> **БИЛЛ ВАГНЕР**
>
> Тот факт, что ноль является значением по умолчанию для переменных перечислимого типа, предполагает, что вы всегда должны быть уверены, что ноль является допустимым элементом любого перечисления.

Значением по умолчанию для любого перечисления типа является целое значение ноль, приведенное к перечислимому типу. В случаях, когда переменные автоматически инициализируются значением по умолчанию, переменным перечисления присваивается это значение. Для того чтобы значение по умолчанию перечисления было легко применять, литерал 0 неявно приводится к любому перечислимому типу. Таким образом, разрешен следующий код:

```
Color c = 0;
```

> **БРЭД АБРАМС**
>
> Мой первый учебный курс по программированию в высшей школе был по Turbo Pascal. (Спасибо, Андерс!) На одном из моих первых заданий, которое мне вернул преподаватель, я увидел большой красный кружок вокруг цифры 65 в исходном коде и небрежную надпись: «Никаких магических констант!!». Мой учитель в то время объяснял мне преимущества использования константы `RetirementAge` для удобного чтения и поддержки. Перечисления делают это решение суперлегким. В отличие от других языков программирования, использование перечислений в C# не вызывает потери производительности во время выполнения. Хотя я видел множество оговорок в обзорах API, я не нахожу достаточных причин использовать вместо перечислений магические константы!

1.11. Делегаты

Делегат представляет собой ссылки на методы с заданным списком параметров и типом результата. Делегаты дают возможность использовать методы как сущности, которые могут быть присвоены переменным или передаваться как параметры. Понятие делегатов подобно понятию указателей на функции в некоторых других языках, но в отличие от указателей делегаты являются объектно-ориентированными и поддерживают безопасность типов.

В следующем примере описан и используется тип делегата, именуемый `Function`.

```
using System;
delegate double Function(double x);
class Multiplier
{
    double factor;
    public Multiplier(double factor) {
        this.factor = factor;
    }
    public double Multiply(double x) {
```

продолжение ⇗

```
        return x * factor;
    }
}
class Test
{
    static double Square(double x) {
        return x * x;
    }
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

Экземпляр делегата `Function` может ссылаться на любой метод, который в качестве аргумента принимает `double` и возвращает значение `double`. Метод `Apply` применяет `Function` к элементам `double[]`, возвращая `double[]` с результатами. В методе `Main` `Apply` используется для применения трех различных функций к `double[]`.

Делегаты могут ссылаться либо на статические методы (такие как `Square` или `Math.Sin` в предыдущем примере), либо на методы экземпляра (такие как `Multiply` в предыдущем примере). Делегаты, которые ссылаются на метод экземпляра, ссылаются также на отдельный объект, и когда делегат вызывает метод экземпляра, `this` принимает значение этого объекта.

Делегаты также могут создаваться с помощью анонимных функций, которые представляют собой «встроенные методы», так как они создаются «на лету». Анонимные функции могут видеть локальные переменные окружающих методов. Таким образом, вышеизложенный пример с множителем может быть более просто записан без использования класса `Multiplier`:

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

БИЛЛ ВАГНЕР

Это свойство делегатов делает их отличным инструментом для обеспечения интерфейса между компонентами, очень слабо связанными между собой.

1.12. Атрибуты

Типы, элементы и другие сущности в программах на C# могут содержать модификаторы, которые управляют определенными аспектами их поведения. Например, доступностью метода можно управлять с помощью модификаторов `public`,

`protected`, `internal` и `private`. В C# эта возможность получила развитие: определяемые пользователем типы описательной информации можно присоединить к программным объектам и извлечь из среды выполнения. В программах эта дополнительная описательная информация задается с помощью *атрибутов*.

В следующем примере описан атрибут `HelpAttribute`, который может быть связан с некоторой сущностью в программе, чтобы привязать эту сущность к соответствующей документации.

```
using System;
public class HelpAttribute: Attribute
{
    string url;
    string topic;
    public HelpAttribute(string url) {
        this.url = url;
    }
    public string Url {
        get { return url; }
    }
    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

Все классы атрибутов происходят от базового класса `System.Attribute`, описанного в .NET Framework. Атрибуты применяют путем указания их имени вместе с аргументами, записанными в квадратных скобках, непосредственно перед соответствующим описанием. Если имя атрибута оканчивается на `Attribute`, эта часть имени при ссылке на атрибут может быть опущена. Например, атрибут `HelpAttribute` может использоваться следующим образом:

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) { }
}
```

В этом примере `HelpAttribute` связывается с классом `Widget`, а другой `HelpAttribute` — с методом `Display` этого класса. Открытые конструкторы класса-атрибута управляют информацией, которая должна предоставляться, когда атрибут связывается с программным объектом. Добавочная информация может быть предоставлена с помощью ссылки на открытые свойства чтения-записи класса-атрибута (такой как ссылка на свойство `Topic` в предыдущем примере).

Следующий пример показывает, как можно извлечь информацию из атрибута данного объекта программы во время выполнения программы с помощью механизма рефлексии.

```
using System;
using System.Reflection;
class Test
```

продолжение ☞

```
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine(" Url={0}, Topic={1}",
                                a.Url, a.Topic);
        }
    }
    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

Когда отдельные атрибуты запрашиваются с помощью рефлексии, вызывается конструктор для атрибута класса с информацией, содержащейся в исходном коде программы, и возвращается созданный экземпляр атрибута. Если дополнительная информация обеспечивается через свойства, перед возвратом экземпляра атрибута этим свойствам присваиваются значения.

БИЛЛ ВАГНЕР

Полностью потенциал атрибутов будет реализован, когда будущая версия компилятора C# позволит разработчикам читать атрибуты и использовать их для изменения кода до того, как компилятор создаст IL. Мне хочется использовать атрибуты для изменения поведения кода с момента выпуска первой версии C#.

Глава 2

Лексическая структура

2.1. Программы

Программы, написанные на C#, состоят из одного или нескольких **исходных файлов**, формально называемых **единицами компиляции** (раздел 9.1). Исходный файл является упорядоченной последовательностью символов Unicode. Обычно исходные файлы находятся во взаимно однозначном соответствии с файлами файловой системы, хотя это не является обязательным. Для обеспечения максимальной переносимости рекомендуется кодировать файлы файловой системы с помощью кодировки UTF-8.

При компиляции программа проходит три стадии:

1. Преобразование файла из специфического набора символов и схемы кодирования в последовательность символов Unicode.

2. Лексический анализ, который преобразует поток входящих символов в поток лексем.

3. Синтаксический анализ, который преобразует поток лексем в исполняемый код.

2.2. Грамматики

В этом описании представлен синтаксис языка программирования C#, использующий две грамматики.

Лексическая грамматика (раздел 2.2.2) определяет, каким образом символы Unicode комбинируются для формирования признаков конца строки, пробелов, символов табуляции и пустой строки, комментариев, лексем и директив препроцессора. **Синтаксическая грамматика** (раздел 2.2.3) определяет, каким образом лексемы, полученные по правилам лексической грамматики, комбинируются для формирования программы на C#.

2.2.1. Система обозначений

Лексическая и синтаксическая грамматики используют **грамматические правила**. Каждое грамматическое правило определяет нетерминальный символ или возможное расширение этого нетерминального символа в последовательность

нетерминальных или терминальных символов. В грамматических правилах нетерминальные символы записаны *курсивом*, а терминальные символы — `моноширинным шрифтом`.

Первая строка грамматического правила содержит имя определяемого нетерминального символа, за которым следует двоеточие. Каждая последующая строка с отступом содержит возможное расширение этого нетерминального символа, представленное как последовательность нетерминальных и терминальных символов. Например, правило

```
оператор-while:
   while ( булевское-выражение ) вложенный-оператор
```

определяет *оператор-while*, состоящий из лексемы `while`, за ней следует лексема «`(`», за ней *булевское выражение*, затем «`)`», затем *вложенный-оператор*.

Когда возможных расширений нетерминального символа более одного, альтернативы записываются в отдельных строчках. Например, правило

```
список-операторов:
   оператор
   список-операторов оператор
```

определяет, что *список-операторов* включает в себя либо *оператор*, либо *список-операторов*, за которым следует *оператор*. Иными словами, определение рекурсивно и означает, что список операторов состоит из одного или более операторов.

Подстрочный суффикс «$_{\text{opt}}$» используется для описания необязательного символа. Правило

```
блок:
   { список-операторов_opt }
```

является сокращением

```
блок:
   { }
   { список-операторов }
```

и определяет, что *блок* включает в себя необязательный *список-операторов*, заключенный между лексемами «`{`» и «`}`».

Альтернативные варианты обычно размещаются на отдельных строках, хотя в случаях, когда их много, списку расширений, представленному в одной строке, может предшествовать выражение «одно из». Это просто более короткая запись по сравнению с записью каждой альтернативы в отдельной строке.

Например, правило

```
суффикс-вещественного-типа: одно из
   F f D d M m
```

является краткой записью для

```
суффикс-вещественного-типа:
   F
   f
   D
   d
   M
   m
```

2.2.2. Лексическая грамматика

Лексическая грамматика C# описана в разделах 2.3 — 2.5. Терминальные символы лексической грамматики являются символами Unicode, и она определяет, как комбинируются символы для формирования лексем (раздел 2.4), пробелов, символов табуляции и символов пустой строки (раздел 2.3.3), комментариев (раздел 2.3.2) и директив препроцессора (раздел 2.5).

Каждый исходный файл в программе C# должен соответствовать правилу *исходный-текст* лексической грамматики (раздел 2.3).

2.2.3. Синтаксическая грамматика

Синтаксическая грамматика C# представлена в главах и приложениях, следующих за этой главой. Терминальными символами синтаксической грамматики являются лексемы, определяемые лексической грамматикой. Синтаксическая грамматика определяет, каким образом лексемы комбинируются для формирования программы на C#.

Каждый исходный файл в программе C# должен соответствовать правилу *единица-компиляции* синтаксической грамматики (раздел 9.1).

2.3. Лексический анализ

Правило *исходный-текст* определяет лексическую структуру исходного файла C#. Каждый исходный файл программы C# должен соответствовать этому лексическому грамматическому правилу.

```
исходный-текст:
   секция-исходного-текста opt

секция-исходного-текста:
   часть-секции-исходного-текста
   секция-исходного-текста   часть-секции-исходного-текста

часть-секции-исходного-текста:
   входные-элементыopt   новая-строка
   директива-препроцессора

входные-элементы:
   входной-элемент
   входные-элементы   входной-элемент

входной-элемент:
   пробельный-символ
   комментарий
   лексема
```

Пять базовых элементов составляют лексическую структуру исходного файла C#: символы конца строки (см. раздел 2.3.1), пробельные символы (раздел 2.3.2),

лексемы (раздел 2.4) и директивы препроцессора (раздел 2.5). Из этих базовых элементов только лексемы являются значимыми для синтаксической грамматики программы (раздел 2.2.3).

Лексическая обработка исходного файла C# состоит в преобразовании файла в последовательность лексем, которые затем становятся исходными данными для синтаксического анализа. Для разделения лексем служат символы конца строки, пробельные символы и комментарии. Результатом действия директив препроцессора может быть пропуск некоторых секций исходного файла, в остальных случаях эти лексические элементы не оказывают воздействия на синтаксическую структуру программы C#.

Когда последовательности символов в исходном файле соответствует несколько лексических правил, всегда формируется наиболее длинный из возможных лексических элементов. Например, последовательность символов `//` обрабатывается как начало однострочного комментария, поскольку этот лексический элемент длинней, чем единственный символ `/`.

2.3.1. Символы конца строки

Символы конца строки разделяют символы исходного файла C# на строки.

```
новая-строка:
   Символ возврата каретки (Carriage return character) (U+000D)
   Символ перевода строки (Line feed character) (U+000A)
   Символ возврата каретки (Carriage return character) (U+000D),
       за которым следует символ перевода строки (line feed character) (U+000A)
   Символ новой строки (Next line character) (U+0085)
   Символ разделения строк (Line separator character) (U+2028)
   Символ разделения абзацев (Paragraph separator character) (U+2029)
```

Для совместимости с исходным кодом средств редактирования, которые добавляют маркеры конца файла, и для обеспечения возможности просмотра исходного файла в виде последовательности строк с каждым исходным файлом программы C# выполняются по порядку следующие преобразования:

- Если последний символ исходного файла есть Control-Z (U+001A), этот символ удаляется.
- Символ возврата каретки (U+000D) добавляется к концу исходного файла, если исходный файл не пуст и если последний символ исходного файла не является символом возврата каретки (U+000D), символом протяжки бумаги (U+000A), разделителем строк (U+2028) или разделителем абзацев (U+2029).

2.3.2. Комментарии

Поддерживается два вида комментариев: однострочные и многострочные. **Однострочные комментарии** начинаются с символов `//` и продолжаются до конца строки. **Многострочные комментарии** начинаются с символов `/*` и заканчиваются символами `*/`. Эти комментарии могут занимать несколько строк.

```
комментарий:
   однострочный-комментарий
   многострочный-комментарий

однострочный-комментарий:
   //   входные-символыopt

входные-символы:
   входной-символ
   входные-символы   входной-символ

входной-символ:
   Любой символ Unicode, кроме символа-новой-строки

символ-новой-строки:
   Символ возврата каретки (U+000D)
   Символ протяжки бумаги (U+000A)
   Символ новой строки (U+0085)
   Символ разделения строк (U+2028)
   Символ разделения абзацев (U+2029)

многострочный-комментарий:
   /*   текст-многострочного-комментарияopt   звездочки /

текст-многострочного-комментария:
   секция-многострочного-комментария
   текст-многострочного-комментария   секция-многострочного-комментария

секция-многострочного-комментария:
   /
   звездочкиopt   не-слэш-или-звездочка

звездочки:
   *
   звездочки *

не-слэш-или-звездочка:
   Любой символ Unicode, кроме / или *
```

Комментарии не могут быть вложенными. Последовательности символов `/*` и `*/` не имеют специального значения внутри комментария, начинающегося с `//`, а последовательности символов `//` и `/*` не имеют специального значения внутри многострочного комментария.

Комментарии в символьных и строковых литералах не обрабатываются.

Пример

```
/*Программа Здравствуй, мир
Эта программа выводит текст "здравствуй, мир" на консоль
*/
class Hello
{
   static void Main() {
      System.Console.WriteLine("здравствуй, мир");
   }
}
```

содержит комментарий с разделителями.

Пример

```
// Программа Здравствуй, мир
// Эта программа выводит текст "здравствуй, мир" на консоль
//
class Hello                  // этому классу можно дать любое имя
{
   static void Main() {    // этот метод должен именоваться "Main"
      System.Console.WriteLine("здравствуй, мир");
   }
}
```

содержит несколько однострочных комментариев.

2.3.3. Пробельные символы

Пробельные символы определяются как любой символ Unicode класса Zs (который включает символ пробела), а также символ горизонтальной табуляции, символ вертикальной табуляции и символ новой страницы.

```
пробельный-символ:
   Любой символ Unicode класса Zs
   Символ горизонтальной табуляции (U+0009)
   Символ вертикальной табуляции (U+000B)
   Символ новой страницы (U+000C)
```

2.4. Лексемы

Существуют следующие виды лексем: идентификаторы, ключевые слова, литералы, операции и знаки пунктуации. Пробельные символы и комментарии лексемами не являются, они служат разделителями для лексем.

```
лексема:
   идентификатор
   ключевое-слово
   целочисленный-литерал
   вещественный-литерал
   символьный-литерал
   строковый-литерал
   знак-операции-или-пунктуации
```

2.4.1. Управляющие последовательности Unicode

Управляющая последовательность Unicode представляет собой символ Unicode. Он обрабатывается в идентификаторах (раздел 2.4.2), символьных константах (раздел 2.4.4.4) и обычных строковых константах (раздел 2.4.4.5). В любом другом месте управляющие последовательности Unicode не обрабатываются (например, при формировании операций, знаков пунктуации или ключевых слов).

```
управляющая-последовательность-unicode:
    \u шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра
                         шестнадцатеричная-цифра
    \U шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-
                         цифра шестнадцатеричная-цифра шестнадцатеричная-цифра
                         шестнадцатеричная-цифра
                         шестнадцатеричная-цифра шестнадцатеричная-цифра
```

ЭРИК ЛИППЕРТ

Эта особенность отличает C# от Java, в котором управляющие последовательности Unicode могут появляться почти везде.

Управляющая последовательность Unicode представляет собой единственный символ Unicode, образуемый шестнадцатеричным числом, следующим за символами `\u` или `\U`. Поскольку в C# используется 16-разрядное кодирование символов и строк в кодировке Unicode, символы Unicode в интервале от U+10000 до U+10FFFF в символьных константах не разрешены, а в строковых константах заменяются парой Unicode-символов-заместителей. Символы Unicode с кодами свыше 0x10FFFF не поддерживаются.

Множественные подстановки не поддерживаются. Например, строковая константа \u005Cu005C эквивалентна \u005C, а не `\`. Символ `\` в кодировке Unicode имеет значение \u005C.

Пример

```
class Class1
{
   static void Test(bool \u0066) {
      char c = '\u0066';
      if (\u0066)
         System.Console.WriteLine(c.ToString());
   }
}
```

демонстрирует варианты использования символа \u0066, который представляет собой управляющую последовательность для буквы «f». Программа эквивалентна

```
class Class1
{
   static void Test(bool f) {
      char c = 'f';
      if (f)
         System.Console.WriteLine(c.ToString());
   }
}
```

2.4.2. Идентификаторы

Правила для идентификаторов, приведенные в этом разделе, в точности соответствуют рекомендованным в Unicode Standard Annex 31, за исключением того, что: подчеркивание разрешено в качестве начального символа (как это

традиционно принято в языках программирования C), управляющие последовательности Unicode разрешены в идентификаторах, а символы `@` разрешены в качестве префикса, который дает возможность использовать ключевые слова как идентификаторы.

идентификатор:
   доступный-идентификатор
   `@` *идентификатор-или-ключевое-слово*

доступный-идентификатор:
   идентификатор-или-ключевое-слово, не являющийся *ключевым-словом*

идентификатор или-ключевое-слово:
   символ-начала-идентификатора   *символы-идентификатора*$_{opt}$

символ-начала-идентификатора:
   буквенный-символ
   `_` (символ подчеркивания U+005F)

символы-идентификатора:
   символ-идентификатора
   символы-идентификатора   *символ-идентификатора*

символ-идентификатора:
   буквенный-символ
   десятичная-цифра
   символ-соединения
   символ-комбинирования
   символ-управления-форматом

буквенный-символ:
   Символ Unicode классов Lu, Ll, Lt, Lm, Lo и Nl
   управляющая-последовательность-unicode,
      представляющая символ классов Lu, Ll, Lt, Lm, Lo и Nl

символ-комбинирования:
   Символ Unicode классов Mn или Mc
   управляющая-последовательность-unicode,
      представляющая символ классов Mn или Mc

десятичная-цифра:
   символ Unicode класса Nd
   управляющая-последовательность-unicode, представляющая символ класса Nd

символ-соединения:
   символ Unicode класса Pc
   управляющая-последовательность-unicode, представляющая символ класса Pc

символ-управления-форматом:
   Символ Unicode класса Cf
   управляющая-последовательность-unicode, представляющая символ класса Cf

Для получения информации относительно классов символов Unicode, упоминавшихся выше, см. Стандарт Unicode версии 3.0 (*The Unicode Standard, Version 3.0*), раздел 4.5.

Примеры допустимых идентификаторов: `identifier1`, `_identifier2`, `@if`.

Идентификаторы в программе, соответствующей стандарту, должны иметь канонический формат, определяемый формой нормализации C (Unicode Normalization Form C), как задано в Unicode Standard Annex 15. Действия при обнаружении идентификатора, которого нет в Normalization Form C, определяются реализацией; диагностика не требуется.

Префикс `@` дает возможность использовать идентификаторы в качестве ключевых слов, что бывает полезно при взаимодействии с другими языками программирования. Символ `@` не является в действительности частью идентификатора, так что в других языках идентификатор может рассматриваться как обычный идентификатор без префикса. Идентификатор с префиксом `@` называется *буквальным идентификатором*. В идентификаторах, которые не являются ключевыми словами, использование префикса `@` разрешено, но крайне не рекомендуется как плохой стиль.

В примере

```
class @class
{
   public static void @static(bool @bool) {
      if (@bool)
         System.Console.WriteLine("true");
      else
         System.Console.WriteLine("false");
   }
}

class Class1
{
   static void M() {
      cl\u0061ss.st\u0061tic(true);
   }
}
```

определяется класс, названный `class` со статическим методом, названном `static`, который имеет параметр `bool`. Заметим, что, так как управляющие символы Unicode не разрешены в ключевых словах, cl\u0061ss есть идентификатор, причем такой же, как `@class`.

Идентификаторы считаются одинаковыми, если они идентичны после проведения следующих преобразований:

- Префикс `@`, если он использовался, удаляется.
- Каждая *управляющая-последовательность-Unicode* преобразуется в соответствующий символ Unicode.
- Все символы форматирования удаляются.

Идентификаторы, содержащие два последовательных символа подчеркивания (U+005F), зарезервированы для использования в конкретной реализации. Например, реализация может обеспечить расширенные ключевые слова, начинающиеся с двух подчеркиваний.

2.4.3. Ключевые слова

Ключевые слова представляют собой подобные идентификаторам последовательности символов, которые зарезервированы и не могут использоваться в качестве идентификаторов, за исключением случая, когда их предваряет символ `@`.

```
ключевое-слово: одно из
   abstract     as         base         bool        break
   byte         case       catch        char        checked
   class        const      continue     decimal     default
   delegate     do         double       else        enum
   event        explicit   extern       false       finally
   fixed        float      for          foreach     goto
   if           implicit   in           int         interface
   internal     is         lock         long        namespace
   new          null       object       operator    out
   override     params     private      protected   public
   readonly     ref        return       sbyte       sealed
   short        sizeof     stackalloc   static      string
   struct       switch     this         throw       true
   try          typeof     uint         ulong       unchecked
   unsafe       ushort     using        virtual     void
   volatile     while
```

В некоторых случаях в грамматике отдельные идентификаторы имеют специальное значение, но не являются ключевыми словами. Такие идентификаторы иногда называют «контекстными ключевыми словами». Например, в объявлении свойства идентификаторы `get` и `set` имеют специальное значение (раздел 10.7.2). Другие идентификаторы, кроме `get` или `set`, не могут находиться в этих позициях, так что конфликт при использовании этих слов в качестве идентификатора не возникает. В других случаях, например при неявном задании типа локальной переменной с идентификатором `var` (раздел 8.5.1), контекстные ключевые слова могут конфликтовать с объявленными именами. В таких случаях объявленное имя имеет преимущество перед идентификатором, используемым в качестве контекстного слова.

ЭРИК ЛИППЕРТ

Новые зарезервированные ключевые слова не добавляются в C# с момента выпуска первой версии. Все новые свойства языка, которые требуют новых ключевых слов (`yield`, `select` и т. д.), используют «контекстные ключевые слова», которые не зарезервированы и имеют специальное значение только в контексте. Такое решение сохраняет обратную совместимость с существующими программами.

ВЛАДИМИР РЕШЕТНИКОВ

Если вам нужен полный перечень контекстных ключевых слов (включая поддерживаемые целевые объекты атрибута), вот он:

```
add alias ascending assembly by descending dynamic equals field from get
global group into join let method module on orderby param partial property
remove select set type typevar value var where yield
```

2.4.4. Литералы (константы)

Литерал является представлением значения в исходном коде.

```
литерал:
   булевский-литерал
   целочисленный-литерал
   вещественный-литерал
   символьный-литерал
   строковый-литерал
   нулевой-литерал
```

2.4.4.1. Булевские константы

Существует два значения булевских констант: `true` и `false`.

```
булевский-литерал:
   true
   false
```

Булевские константы имеют тип `bool`.

2.4.4.2. Целочисленные константы

Целочисленные константы используются для записи значений типов `int`, `uint`, `long` и `ulong`. Они могут иметь две формы записи: десятичную и шестнадцатеричную.

```
целочисленный-литерал:
   десятичный-целочисленный-литерал
   шестнадцатеричный-целочисленный-литерал

десятичный-целочисленный-литерал:
   десятичные-цифры   суффикс-целого-типа_opt

десятичные-цифры:
   десятичная-цифра
   десятичные-цифры   десятичная-цифра

десятичные-цифры: одно из
   0   1   2   3   4   5   6   7   8   9

суффикс-целого-типа: одно из
   U   u   L   l   UL   Ul   uL   ul   LU   Lu   lU   lu

шестнадцатеричный-целочисленный-литерал:
   0x     шестнадцатеричные-цифры       суффикс-целого-типа_opt
   0X     шестнадцатеричные-цифры       суффикс-целого-типа_opt

шестнадцатеричные-цифры:
   шестнадцатеричная-цифра
   шестнадцатеричные-цифры      шестнадцатеричная-цифра

шестнадцатеричная-цифра: одно из
   0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
```

> **ЭРИК ЛИППЕРТ**
>
> C# не поддерживает восьмеричные константы по двум причинам. Во-первых, вряд ли кто-нибудь в наши дни их использует. Во-вторых, если поддерживать их в стандартном формате «ведущий ноль означает восьмеричный формат», это будет потенциальным источником ошибок. Рассмотрим следующий код:
>
> ```
> FlightNumber = 0541;
> ```
>
> Ясно, что в этом выражении имеется в виду десятичная, а не восьмеричная константа.

Типы целочисленных констант определяются следующим образом:

- Если у константы нет суффикса, она имеет первый из типов `int`, `uint`, `long`, `ulong`, в котором ее значение может быть представлено.
- Если у константы есть суффикс `U` или `u`, она имеет первый из типов `uint`, `ulong`, в котором ее значение может быть представлено.
- Если у константы есть суффикс `L` или `l`, она имеет первый из типов `long`, `ulong`, в котором ее значение может быть представлено.
- Если у константы есть суффикс `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, или `lu`, ее тип `ulong`.

Если значение целочисленной константы выходит за рамки интервала значений типа `ulong`, выдается ошибка компиляции.

Если говорить о стиле, при написании константы типа `long` предпочтительнее использовать `L` вместо `l`, так как букву «l» легко перепутать с цифрой «1».

Чтобы предотвратить малейшую возможность записать значения типов `int` и `long` в виде десятичных целых констант, существуют следующие два правила:

- Если *десятичный-целочисленный-литерал* со значением 2147483648 (2^{31}) без *суффикса-целого-типа* непосредственно следует за лексемой унарной операции минус (раздел 7.7.2), результатом является константа типа `int` со значением –2147483648 (-2^{31}). Во всех остальных случаях такой *десятичный-целочисленный-литерал* имеет тип `uint`.
- Если *десятичный-целочисленный-литерал* со значением 9223372036854775808 (2^{63}) без *суффикса-целого-типа* или *с суффиксом-целого-типа* `L` или `l` непосредственно следует за лексемой унарной операции минус (раздел 7.7.2), результатом является константа типа `long` со значением –9223372036854775808 (-2^{63}). Во всех остальных случаях такой *десятичный-целочисленный-литерал* имеет тип `ulong`.

> **ДЖОЗЕФ АЛЬБАХАРИ**
>
> Благодаря неявным преобразованиям константных выражений (раздел 6.1.8) целые константы могут быть непосредственно присвоены переменным типа `uint`, `long` и `ulong` (а также `short`, `ushort`, `byte` и `sbyte`):
>
> ```
> uint x = 3; long y = 3; ulong z = 3;
> ```
>
> Вследствие этого в суффиксах U и L редко возникает необходимость. Пример, когда они все же полезны, — явное задание 64-разрядных вычислений над константами, для которых в противном случае применялась бы 32-разрядная арифметика:

```
long error = 1000000 * 1000000 ;     // Ошибка компиляции (32-разрядное
                                     // переполнение)
long trillion = 1000000L * 1000000L; // Правильно (переполнения нет)
```

2.4.4.3. Вещественные константы

Вещественные константы используются для записи значений типов `float`, `double` и `decimal`.

```
вещественный-литерал:
   десятичные-цифры  .  десятичные-цифры   экспоненциальная-частьopt
       суффикс-вещественного-типаopt
   .  десятичные цифры   экспоненциальная часть opt   суффикс-вещественного-типаopt
   десятичные-цифры   экспоненциальная-часть   суффикс-вещественного-типаopt
   десятичные-цифры   суффикс-вещественного-типа

экспоненциальная-часть:
   e знак opt  десятичные-цифры
   E знакopt десятичные- цифры

знак: одно из
   +    -

суффикс-вещественног-типа: одно из
   F   f   D   d   M   m
```

Если *суффикс-вещественного-типа* отсутствует, тип вещественной константы — `double`. В противном случае суффикс вещественного типа определяет тип вещественной константы следующим образом:

- Вещественные константы с суффиксом `F` или `f` имеют тип `float`. Например, константы `1f`, `1.5f`, `1e10f` и `123.456F` имеют тип `float`.
- Вещественные константы с суффиксом `D` или `d` имеют тип `double`. Например, константы `1d`, `1.5d`, `1e10d` и `123.456D` имеют тип `double`.
- Вещественные константы с суффиксом M или m имеют тип `decimal`. Например, константы `1m`, `1.5m`, `1e10m` и `123.456M` имеют тип `decimal`. Когда константа приводится к значению типа `decimal`, берется точное значение и при необходимости округляется до ближайшего представимого значения по правилам банковского округления (раздел 4.1.7). Степень константы сохраняется, если ее значение не округляется или не равно нулю (в последнем случае знак и степень будут равны 0). Таким образом, например, преобразование константы `2.900m` дает десятичное значение со знаком 0, коэффициентом 2900 и степенью 3.

Если константу нельзя представить в указанном виде, выдается ошибка компиляции.

Значение вещественной константы типа `float` или `double` определяется при помощи «округления до ближайшего значения» в соответствии со стандартом IEEE.

Заметим, что в вещественной константе после запятой всегда должны быть десятичные цифры. Например, `1.3F` является десятичной константой, а `1.F` не является.

> **ДЖОЗЕФ АЛЬБАХАРИ**
>
> Из всех числовых суффиксов наиболее полезны m и f. Без этих суффиксов дроби с плавающей точкой (float) или десятичные константы (decimal) не могут быть определены без приведения. Например, следующий код не может быть скомпилирован, поскольку константа 1.5 будет воспринята как имеющая тип double:
>
> ```
> float x = 1.5;  // Ошибка: no implicit conversion from double to float
>                 // (отсутствует
>                 // неявное преобразование из double к float
> decimal y = 1.5; // Ошибка: no implicit conversion from double to decimal
>                  // (отсутствует неявное преобразование из double к decimal
> ```
>
> Интересно, что приведенный ниже код *будет* скомпилирован, поскольку в C# определено неявное преобразование от int к decimal:
>
> ```
> decimal z = 123; // Правильно: константа определена как int, а затем неявно
>                  // преобразована к decimal
> ```
>
> Суффикс d является излишним, так как десятичная точка выполняет ту же самую функцию:
>
> ```
> Console.WriteLine ((123.0).GetType() == typeof (double));   // True
> ```

2.4.4.4. Символьные константы

Символьная константа представляет собой единственный символ, и обычно это символ в кавычках, например 'a'.

символьный-литерал:
  ' *символ* **'**

символ:
  единственный-символ
  простая-управляющая-последовательность
  шестнадцатеричная-управляющая-последовательность
  управляющая-последовательность-unicode

единственный-символ:
  Любой символ, кроме ' (U+0027), \ (U+005C) и *символа-новой-строки*

простая-управляющая-последовательность: одно из
  \' \" \\ \0 \a \b \f \n \r \t \v

шестнадцатеричная-управляющая-последовательность:
  \x *шестнадцатеричная-цифра* *шестнадцатеричная-цифра*$_{opt}$
  шестнадцатеричная-цифра$_{opt}$ *шестнадцатеричная-цифра*$_{opt}$

Символ, который следует за символом «обратный слэш» (\), должен быть одним из следующих символов: ', «, \, 0, a, b, f, n, r, t, u, U, x, v. В противном случае выдается ошибка компиляции.

Шестнадцатеричная управляющая последовательность представляет собой единственный символ Unicode со значением, образуемым шестнадцатеричным числом, следующим за управляющим символом \x.

Если значение, представленное символьной константой, больше чем U+FFFF, выдается ошибка компиляции.

Управляющая последовательность символов Unicode (раздел 2.4.1) в символьной константе должна попадать в интервал от U+0000 до U+FFFF.

Простая управляющая последовательность представляет собой кодирование символами Unicode, как это описано в таблице, приведенной ниже.

Управляющая последовательность	**Имя символа**	**Код Unicode**
\’	Одинарная кавычка	0x0027
\»	Двойная кавычка	0x0022
\\	Обратный слэш	0x005C
\0	Нуль-символ	0x0000
\a	Звуковой сигнал	0x0007
\b	«Забой»	0x0008
\f	Протяжка бумаги	0x000C
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальная табуляция	0x0009
\v	Вертикальная табуляция	0x000B

Символьные константы имеют тип `char`.

2.4.4.5. Строковые константы

C# поддерживает две формы строковых констант: *обычные строковые константы* и *буквальные (дословные) строковые константы*.

Обычная строковая константа состоит из нуля и более символов, заключенных в двойные кавычки, например `«hello»`, и может содержать простые управляющие последовательности (например, `\t` для символа табуляции), шестнадцатеричные управляющие последовательности и управляющие последовательности Unicode.

Буквальная строковая константа начинается с символа `@`, за которым следует двойная кавычка, затем ноль и более символов и закрывающая двойная кавычка. В буквальных строковых константах символы между разделителями интерпретируются буквально, единственное исключение составляет *управляющая-последовательность-кавычки*. В частности, простые, шестнадцатеричные и Unicode управляющие последовательности в буквальных строковых константах не обрабатываются. Буквальная строковая константа может занимать несколько строк.

> **ДЖОН СКИТ**
>
> Одно свойство, присущее буквальным строковым константам, которое беспокоит меня, — способ, которым представлен перевод строки в файле. Это, конечно, естественно, но это означает, что изменение в файле концов строк с `\r\n` на `\n` не является чисто декоративным: оно может повлиять на поведение.

```
строковый-литерал:
   обычный-строковый-литерал
   буквальный-строковый-литерал
```

продолжение ☞

```
обычный-строковый-литерал:
   "   символы-обычного-строкового-литерала_opt   "

символы-обычного-строкового-литерала:
   символ-обычного-строкового-литерала
   символы-обычного-строкового-литерала  символ-обычного-строкового-литерала

символ-обычного-строкового-литерала:
   одиночный-символ-обычного-строкового-литерала
   простая-управляющая-последовательность
   шестнадцатеричная-управляющая-последовательность
   управляющая-последовательность-unicode

одиночный-символ-обычного-строкового-литерала:
   Любой символ, за исключением " (U+0022), \ (U+005C) и символа новой строки

буквальный-строковый-литерал:
   @"   символы-буквального-строкового-литерала_opt   "

символы-буквального-строкового-литерала:
     символ-буквального-строкового-литерала
     символы-буквального-строкового-литерала символ-буквального-строкового-литерала

символ-буквального-строкового-литерала:
   одиночный-символ-буквального-строкового-литерала
   управляющая-последовательность-кавычки

одиночный-символ-буквального-строкового-литерала:
   Любой символ за исключением "

управляющая-последовательность-кавычки:
   ""
```

За символом «обратный слэш» (\) в обычной строковой константе может идти один из следующих символов: ‘, «, \, 0, a, b, f, n, r, t, u, U, x, v. В противном случае выдается ошибка компиляции.

Пример

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";                 // hello, world

string c = "hello \t world";                // hello    world
string d = @"hello \t world";               // hello \t world

string e = "Joe said \"Hello\" to me";       // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";      // Joe said "Hello" to me
string g = "\\\\server\\share\\file.txt";    // \\server\share\file.txt
string h = @"\\server\share\file.txt";       // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

демонстрирует разнообразие строковых констант. Последняя строковая константа j является буквальной строковой константой, которая занимает несколько строк.

Символы, расположенные внутри кавычек, включая пробельные символы, такие как перевод строки, воспринимаются буквально, то есть без преобразования.

БРЭД АБРАМС

На заре создания языка C# мы экспериментировали с использованием символа «левая обратная кавычка» (`). Лично мне нравился этот выбор: этот символ относится к группе кавычек и мало используется. Причем он используется настолько редко, что некоторые клавиатуры даже не имеют этой клавиши. Нам повезло, что вовремя нашлись последователи C# во всем мире, что позволило заметить эту потенциальную ошибку на самой ранней стадии создания языка.

Так как шестнадцатеричная управляющая последовательность может иметь различное число шестнадцатеричных цифр, строковая константа «`\x123`» содержит единственный символ с шестнадцатеричным значением 123. Чтобы создать строку, содержащую символ с шестнадцатеричным значением 12, за которым следует 3, нужно написать «`\x00123`» или «`\x12`» + «`3`».

ДЖОН СКИТ

Шестнадцатеричные управляющие последовательности не только редкость, но еще и зло. Хотя такая строка, как «`\x9Tabbed`», достаточно ясна, это все же не так очевидно, как то, что «`\x9Badly tabbed`» действительно начинается с символа Unicode U+9BAD.

Строковые константы имеют тип `string`.

Каждая строковая константа не обязательно представляется в виде отдельного экземпляра строки. Когда в программе появляются две или более строковых константы, эквивалентные в соответствии с операцией равенства строк (раздел 7.10.7), они ссылаются на один и тот же экземпляр строки. Например, вывод, производимый в результате выполнения кода:

```
class Test
{
   static void Main() {
      object a = "hello";
      object b = "hello";
   System.Console.WriteLine(a == b);
   }
}
```

будет `True`, поскольку две константы ссылаются на один и тот же экземпляр строки.

ДЖОЗЕФ АЛЬБАХАРИ

Такая оптимизация называется *интернированием* (*interning*). Одним из полезных результатов интернирования является сокращение объема скомпилированной сборки, поскольку дублирование строковых констант исключается.

продолжение ➪

> В предыдущем примере `a` и `b` были объявлены как объекты типа `object`, что заставляет производить последующее сравнение с использованием ссылочной семантики. Если объявить `a` и `b` как объекты типа `string`, для сравнения будет использоваться операция сравнения типа `string ==`, результатом которой является значение `true`, если строки имеют одинаковое содержимое, даже если они ссылаются на разные объекты.

2.4.4.6. Константа null

нулевой-литерал:
  `null`

Константа `null` (*нулевой-литерал*) может быть неявно приведена к ссылочному или к обнуляемому типу.

> **ЭРИК ЛИППЕРТ**
> Сама по себе константа `null` как выражение типа не имеет.

2.4.5. Знаки операций и пунктуации

Существует несколько видов знаков операций и пунктуации. Знаки операции используются в выражениях для описания действий, производимых над одним или несколькими операндами. Например, в выражении `a + b` используется знак операции `+` для сложения двух операндов `a` и `b`. Знаки пунктуации служат для группирования и разделения.

```
знак-операции-или--пунктуации: одно из
   {    }    [    ]    (    )    .    ,    :    ;
   +    -    *    /    %    &    |    ^    !    ~
   =    <    >    ?    ??   ::   ++   --   &&   ||
   ->   ==   !=   <=   >=   +=   -=   *=   /=   %=
   &=   |=   ^=   <<   <<=  =>
```

```
сдвиг-вправо:
   >|>
```

```
присваивание-со-сдвигом-вправо:
   >|>=
```

Вертикальная черта в правилах для *сдвига-вправо* и *присваивании-со-сдвигом-вправо* используется для обозначения того, что, в отличие от других правил синтаксической грамматики, символы любого типа (даже пробельные) между лексемами не разрешены. Эти правила предусмотрены для того, чтобы обеспечить правильное управление *списком-параметров-типов* (раздел 10.1.3).

2.5. Директивы препроцессора

> **БИЛЛ ВАГНЕР**
>
> Из этого раздела видно, какое влияние оказывает наследие языка C на современные разработки. Сам я крайне редко использую директивы препроцессора — но я представляю, какой вал критики обрушился бы на C#, если бы в него не входила мощная сеть директив препроцессора.

Директивы препроцессора позволяют при некоторых условиях пропускать разделы исходных файлов, выдавать сообщения об ошибках и предупреждения и точно очерчивать секции исходного кода. Сам термин «директивы препроцессора» используется для обозначения связи с языками C и C++. В C# нет отдельного шага препроцессора: директивы препроцессора обрабатываются на стадии лексического анализа.

```
директива-препроцессора:
    директива-объявления
    условная-директива
    директива-line
    директива-диагностики
    директива-region
    директива-pragma
```

Доступны следующие директивы препроцессора:

- `#define` и `#undef` используются для соответственно определения и отмены определения символов условной компиляции (раздел 2.5.3).
- `#if`, `#elif`, `#else` и `#endif` служат для условного пропуска секций исходного кода (раздел 2.5.4).
- `#line` служит для управления номерами строк, используемыми в предупреждениях и сообщениях об ошибках (раздел 2.5.7).
- `#error` и `#warning` используются соответственно для вывода ошибок и предупреждений (раздел 2.5.5).
- `#region` и `#endregion` используются для явного задания секций исходного кода (раздел 2.5.6).
- `#pragma` используется для определения необязательной контекстной информации для компилятора (раздел 2.5.8).

Директивы препроцессора всегда занимают отдельную строку исходного кода и всегда начинаются с символа `#` и имени директивы. Перед символом # и между символом # и именем директивы препроцессора могут находиться пробельные символы.

Строка исходного кода, содержащая директивы `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif` или `#line`, может оканчиваться однострочным комментарием. Многострочные комментарии (вида `/* */`) в строках исходного кода, содержащего директивы препроцессора, не допускаются.

Директивы препроцессора не являются лексемами и не являются частью синтаксической грамматики C#. Однако они могут использоваться для того, чтобы

включать или исключать последовательность лексем, и таким способом могут влиять на значение программы C#. Например, при компиляции программа

```
#define A
#undef B
class C
{
#if A
   void F() {}
#else
   void G() {}
#endif
#if B
   void H() {}
#else
   void I() {}
#endif
}
```

в результате дает точно такую же последовательность лексем, как программа

```
class C
{
   void F() {}
   void I() {}
}
```

Таким образом, хотя лексически эти программы довольно сильно отличаются, синтаксически они одинаковы.

2.5.1. Символы условной компиляции

Функциональные возможности условной компиляции, обеспечиваемые директивами `#if`, `#elif`, `#else` и `#endif`, управляются при помощи выражений препроцессора (раздел 2.5.2) и символов условной компиляции.

символ-условной-компиляции:
   Любой *идентификатор-или-ключевое-слово* за исключением **true** или **false**

Символы условной компиляции имеют два возможных состояния: *определенное* и *неопределенное*. В начале лексической обработки исходного файла символ условной компиляции не определен до тех пор, пока он не будет явно определен с помощью внешнего механизма (такого как опция компилятора командной строки). Когда обрабатывается директива `#define`, символ условной компиляции, упомянутый в этой директиве, становится определенным для данного исходного файла. Он остается определенным до того момента, как будет обработана директива `#undef` для того же символа или при достижении конца файла. Выполнение директив `#define` и `#undef` в одном исходном файле не оказывает влияния на другие исходные файлы той же программы.

При использовании в выражении препроцессора определенный символ условной компиляции имеет булевское значение `true`, а неопределенный символ — значение `false`. Не требуется, чтобы символы условной компиляции были явно объявлены до их использования в выражении препроцессора. Необъявленные символы просто не определены и, таким образом, имеют значение `false`.

Пространство имен для символов условной компиляции отличается от других именованных сущностей программы C#. На символы условной компиляции можно ссылаться только в директивах `#define` и `#undef` и в выражениях препроцессора.

2.5.2. Выражения препроцессора

Выражения препроцессора могут появляться в директивах `#if` и `#elif`. В выражениях препроцессора разрешены операции `!`, `==`, `!=`, `&&`, и `||`, для группирования могут использоваться скобки.

```
выражение-препроцессора:
    пробельный-символopt   выражение-or   пробельный-символopt

выражение-or:
    выражение-and
    выражение-or  пробельный-символopt  ||  пробельный-символopt   выражение-and

выражение-and:
    выражение-равенства
    выражение-and пробельный-символopt  &&  пробельный-символopt   выражение-равенства

выражение-равенства:
    унарное-выражение-препроцессора
    выражение-равенства     пробельный-символopt == пробельный-символopt
          унарное-выражение-препроцессора
    выражение-равенства     пробельный-символopt != пробельный-символopt
          унарное-выражение-препроцессора

унарное-выражение-препроцессора:
    первичное-выражение-препроцессора
    !   пробельный-символopt   унарное-выражение-препроцессора

первичное-выражение-препроцессора:
    true
    false
    символ-условной-компиляции
    (   пробельный-символopt   выражение-препроцессора   пробельный-символopt   )
```

При использовании в выражении препроцессора определенный символ условной компиляции имеет булевское значение `true`, а неопределенный символ — значение `false`.

Вычисление выражения препроцессора всегда дает в результате булевское значение. Правила вычисления выражений препроцессора такие же, как для константных выражений (раздел 7.19), за исключением того, что единственные определенные пользователем сущности, на которые можно ссылаться, — это символы условной компиляции.

2.5.3. Директивы объявлений

Директивы объявлений используются для задания определенных и неопределенных символов условной компиляции

директива-объявления:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **define** *пробельный-символ*
    условный-символ *директива-новой-строки*
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **undef** *пробельный-символ*
    условный-символ *директива-новой-строки*

директива-новой-строки:
  пробельный-символ$_{opt}$ *однострочный-комментарий*$_{opt}$ *новая-строка*

Обработка директивы #define приводит к тому, что данный символ условной компиляции становится определенным, начиная со строки исходного файла, следующей за директивой. Аналогично, обработка директивы #undef приводит к тому, что данный символ условной компиляции становится неопределенным, начиная со строки исходного файла, следующей за директивой.

Любые директивы #define и #undef в исходном файле должны появляться до первой лексемы (раздел 2.4); в противном случае выдается ошибка компиляции. Интуитивно понятно, что директивы #define и #undef должны предшествовать любому «реальному коду» в исходном файле.

Пример

```
#define Enterprise
#if Professional || Enterprise
   #define Advanced
#endif
namespace Megacorp.Data
{
   #if Advanced
   class PivotTable {...}
   #endif
}
```

является примером допустимого кода, поскольку директивы #define предшествуют первой лексеме (ключевое слово namespace) в исходном файле.

Следующий пример даст в результате ошибку компиляции, поскольку #define следует за реальным кодом:

```
#define A
namespace N
{
   #define B
   #if B
   class Class1 {}
   #endif
}
```

Директива #define может определить символ условной компиляции всегда, без промежуточных директив #undef для этого символа. В примере, приведенном ниже, определяется символ условной компиляции A, а затем он определяется снова:

```
#define A
#define A
```

Директива #undef может отменять определение символа условной компиляции, который не был определен. В примере, приведенном ниже, определяется символ

условной компиляции A, и затем определение отменяется дважды; хотя вторая отмена не имеет результата, она является допустимой.

```
#define A
#undef A
#undef A
```

2.5.4. Директивы условной компиляции

Директивы условной компиляции используются для того, чтобы по условию включать или исключать секции исходного файла.

условная-директива:
  секция-if-директивы *секции-elif-директивы*$_{opt}$ *секция-else-директивы*$_{opt}$
    секция-endif-директивы

секция-if-директивы:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **if** *пробельный-символ*
    выражение-препроцессора *директива-новой-строки*
  условная-секция$_{opt}$

секции-elif-директивы:
  секция-elif-директивы
  секции-elif-директивы *секция-elif-директивы*

секция-elif-директивы :
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **elif** *пробельный-символ*
    выражение-препроцессора *директива-новой-строки*
  условная-секция$_{opt}$

секция-else-директивы:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **else**
    директива-новой-строки *условная-секция*$_{opt}$

секция-endif:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **endif**
    директива-новой-строки

условная-секция:
  секция-исходного-текста
  пропущенная-секция

пропущенная-секция:
  часть-пропущенной-секции
  пропущенная-секция *часть-пропущенной-секции*

часть-пропущенной-секции:
  пропущенные-символы$_{opt}$ *новая-строка*
  директива-препроцессора

пропущенные-символы:
  пробельный-символ$_{opt}$ *нечисловой-символ* *входные-символы*$_{opt}$

нечисловой-символ:
  Любой *входной-символ*, кроме **#**

Как отражено в синтаксисе, директивы условной компиляции должны представлять собой набор, включающий в себя, по порядку, директиву `#if`, ноль или более директив `#elif`, ноль или одну директиву `#else` и директиву `#endif`. Между директивами находятся условные секции исходного кода. Каждая секция управляется непосредственно предшествующей ей директивой. Условные секции могут, в свою очередь, содержать вложенные директивы условной компиляции, обеспечивающие формирование полных наборов.

> **КРИС СЕЛЛЗ**
>
> Я могу понять `#if` или `#endif`, но `#elif`? Это шепелявый помощник Санты? Или неформальное сокращение для некоего большого, серого, морщинистого животного? Я бы предпочел заплатить еще за два символа в `#elseif` просто для того, чтобы я действительно мог это запомнить...

Условная-директива препроцессора выбирает по меньшей мере одну из содержащихся в ней *условных-секций* для обычной лексической обработки.

- *Выражения-препроцессора* директив `#if` и `#elif` вычисляются по порядку до тех пор, пока одно из них не будет иметь значение `true`. Если выражение равно `true`, выбирается соответствующая данной директиве *условная-секция*.
- Если все *выражения-препроцессора* имеют значение `false` и если присутствует директива `#else`, выбирается *условная-секция* директивы `#else`.
- Иначе не выбирается ни одна *условная-секция*.

Выбранная *условная-секция*, если она есть, обрабатывается как обычная *секция-исходного-текста*: исходный код, содержащийся в секции, должен удовлетворять лексической грамматике; из исходного кода секции формируются лексемы; директивы препроцессора в секции действуют согласно правилам, описанным ранее.

Оставшиеся секции, если они есть, обрабатываются как *пропущенные-секции*: за исключением директив препроцессора, исходный код секции не обязан удовлетворять грамматике; из исходного кода секции не формируются лексемы; директивы препроцессора должны быть лексически корректны, но они не обрабатываются. Внутри *условной-секции*, которая обрабатывается как *пропущенная-секция*, все вложенные *условные-секции* (содержащиеся во вложенных конструкциях `#if...#endif` и `#region...#endregion`) также обрабатываются как пропущенные.

Следующий пример показывает, каким образом директивы условной компиляции могут быть вложенными:

```
#define Debug            // Включить отладку
#undef Trace             // Выключить трассировку
class PurchaseTransaction
{
   void Commit() {
      #if Debug
         CheckConsistency();
         #if Trace
```

```
            WriteToLog(this.ToString());
         #endif
      #endif
      CommitHelper();
   }
}
```

За исключением директив препроцессора, пропущенный исходный код не подвергается лексическому анализу.

Например, следующий код является допустимым, несмотря на незавершенный комментарий в разделе `#else`.

```
#define Debug            // Включить отладку
class PurchaseTransaction
{
   void Commit() {
      #if Debug
         CheckConsistency();
      #else
         // Делать что-нибудь еще
      #endif
   }
}
```

Заметим, однако, что директивы препроцессора должны быть лексически корректны даже в пропущенных секциях исходного кода.

> **КРИС СЕЛЛЗ**
>
> По возможности надо стремиться избегать вложенных директив препроцессора просто потому, что по умолчанию самый популярный на планете редактор C#, Visual Studio, будет выравнивать их по левому краю текстового файла, что приводит к тому, что отследить вложенность очень трудно. Например,
>
> ```
> #define Debug          // Включить отладку
> #undef Trace           // Выключить трассировку
> class PurchaseTransaction {
>    void Commit() {
> #if Debug
>       CheckConsistency();
> #if Trace
>       WriteToLog(this.ToString());
> #endif
> #endif
>       CommitHelper();
>    }.
> ```

Директивы препроцессора не обрабатываются, если они появляются внутри многострочных входных элементов. Например, программа

```
class Hello
{
   static void Main() {
      System.Console.WriteLine(@"hello,
#if Debug
      World
```

продолжение ☞

```
#else
      Nebraska
#endif
      ");
   }
}
```

в результате выводит:

```
hello,
#if Debug
      world
#else
      Nebraska
#endif
```

В отдельных случаях набор обрабатываемых директив препроцессора может зависеть от вычисления *выражения-препроцессора*. В примере

```
#if X
   /*
#else
   /* */ class Q { }
#endif
```

независимо от того, определено ли `X`, всегда формируется одинаковый поток лексем (`class Q { }`). Если `X` определено, благодаря многострочному комментарию обрабатываются только директивы препроцессора `#if` и `#endif`. Если `X` не определено, систему директив составляют три директивы (`#if`, `#else`, `#endif`).

2.5.5. Директивы диагностики

Директивы диагностики используются для явного формирования сообщений об ошибках и предупреждений, которые выдаются таким же способом, как и другие сообщения об ошибках компиляции и предупреждения.

директива-диагностики:
   пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **error** *сообщение-препроцессора*
   пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **warning** *сообщение-препроцессора*

сообщение-препроцессора:
   новая-строка
   пробельный-символ *символы-исходного-текста*$_{opt}$ *новая-строка*

Пример

```
#warning Code review needed before check-in
#if Debug && Retail
   #error A build can't be both debug and retail
#endif
class Test {...}
```

всегда выводит предупреждение (`"Code review needed before check-in"`) и сообщение об ошибках компиляции (`"A build can't be both debug and retail"`), если определены оба условных символа `Debug` и `Retail` . Заметим, что сообщения препроцессора могут содержать произвольный текст; в частности, они не обяза-

тельно должны содержать только правильные лексемы, как показывает одинарная кавычка в слове `can’t`.

2.5.6. Директивы region

Директивы `#region` предназначены для явного задания границ фрагментов исходного кода.

директива-region:
  начало-фрагмента *условная-секция*$_{opt}$ *конец-фрагмента*

начало-фрагмента:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **region** *сообщение-препроцессора*

конец-фрагмента:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **endregion** *сообщение-препроцессора*

С фрагментом не связано никакое семантическое значение; фрагменты предназначены для того, чтобы программисты или автоматические инструменты могли задавать секции исходного кода. Сообщение, определенное в директивах `#region` или `#endregion`, также не имеет семантического значения: оно служит только для идентификации фрагмента. Парные директивы `#region` и `#endregion` могут иметь разные *сообщения-препроцессора*.

Лексическая обработка для `region`

```
#region
...
#endregion
```

в точности соответствует лексической обработке директивы условной компиляции в форме:

```
#if true
...
#endif
```

2.5.7. Директивы line

Директивы `#line` могут использоваться для изменения номеров строк и имен исходных файлов, которые выдаются компилятором в предупреждениях или сообщениях об ошибках.

Директивы `#line` чаще всего используются в инструментах метапрограммирования, которые создают исходный код C# из некоторого исходного текста.

директива-line:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **line** *пробельный-символ*
    индикатор-строки *новая-строка*

индикатор-строки:
  десятичные-цифры *пробельный-символ* *имя-файла*
  десятичные-цифры
  default
  hidden

продолжение ☞

```
имя-файла:
   "   символы-имени-файла   "

символы-имени-файла:
   символ-имени-файла
   символы-имени-файла   символ-имени-файла

символ-имени-файла:
      Любой входной-символ за исключением "
```

Когда директивы `#line` отсутствуют, компилятор при выводе сообщает истинные номера строк и имена исходных файлов. Когда обрабатывается директива, содержащая *индикатор-строки*, не равный `default`, компилятор считает, что заданный номер (и имя файла, если оно задано) имеет строка, расположенная *после* директивы.

> **ДЖОН СКИТ**
>
> Если вы создаете инструмент, который генерирует код из некоторого другого исходного кода, вы можете подумать, что самый легкий способ — включить директиву `#line` для каждой строки вывода. Однако не все так просто: директива `#line` в буквальной строковой константе будет обрабатываться как часть строки, а не как директива. Так что следующий код почти наверняка не будет работать так, как это предусматривалось:
>
> ```
> #line 5
> Console.WriteLine(@"First line
> #line 6
> Second line");
> ```

Директива `#line default` отменяет действие всех предыдущих директив `#line`. Компилятор сообщает истинную информацию о последовательности строк так, как если бы директив `#line` не было.

Директива `#line hidden` не оказывает влияния на отображаемые в сообщениях об ошибках имена файлов и номера строк, но она влияет на отладку на уровне исходного текста. Когда идет процесс отладки, все строки между директивой `#line hidden` и следующей директивой `#line` (которая не является директивой `#line hidden`) не имеют номеров. При пошаговой отладке отладчик будет пропускать эти строки целиком.

Заметим, что *имя-файла* отличается от обычной строковой константы тем, что управляющий символ не обрабатывается; символ \ внутри *имени-файла* обозначает обычный обратный слэш.

2.5.8. Директива pragma

Директива препроцессора `#pragma` используется для задания необязательной контекстной информации для компилятора. Информация, которая задается в директиве `#pragma`, никоим образом не меняет семантику программы.

директива-pragma:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **pragma** *пробельный-символ* *тело-pragma* *новая-строка*

тело-pragma:
  тело-pragma-warning

C# поддерживает директивы `#pragma` для управления предупреждениями компилятора. Последующие версии языка, возможно, будут включать добавочные директивы `#pragma`. Чтобы обеспечить совместимость с другими компиляторами C#, компилятор Microsoft C# не выдает ошибку компиляции при обнаружении неизвестной директивы `#pragma`; при обнаружении такой директивы, однако, генерируется предупреждение.

2.5.8.1. Предупреждения pragma warning

Директива препроцессора `#pragma warning` используется для отключения или восстановления всех или заданных предупреждений компилятора в следующем за ней тексте программы.

тело-pragma-warning:
  warning *пробельный-символ* *действие-warning*
  warning *пробельный-символ* *действие-warning* *пробельный-символ* *список-warning*

действие-warning:
  disable
  restore

список-warning:
  десятичные-цифры
  список-warning *пробельный-символ*$_{opt}$ **,** *пробельный-символ*$_{opt}$ *десятичные-цифры*

Директива `#pragma warning`, в которой отсутствует список предупреждений, действует на все предупреждения. Директива `#pragma warning`, которая содержит список предупреждений, действует только на те предупреждения, которые определены в этом списке.

> **ДЖОН СКИТ**
>
> Я не уверен, что действие на все предупреждения по умолчанию — хорошая идея. Требование явного `all` как альтернатива списку предупреждений не приводила бы к конфликту с другими предупреждениями (поскольку отдельные предупреждения должны быть численными) и внесло бы ясность. Сказав это, я должен признаться, что никогда не видел, чтобы кто-то отключал все предупреждения, — и надеюсь, что никогда этого не увижу.

Директива `#pragma warning disable` отключает все предупреждения или только заданные.

Директива `#pragma warning restore` восстанавливает все или только заданные предупреждения в состояние, которое они имели в начале единицы компиляции. Заметим, что отдельные предупреждения могут быть отключены внешним спосо-

бом, и `#pragma warning restore` (для всех или для отдельных предупреждений) не сможет восстановить эти предупреждения.

В следующем примере показано применение директивы `#pragma warning` для временного отключения предупреждения, сообщающего о ссылке на устаревшие элементы и использующего номер предупреждения, полученный от компилятора Microsoft C#.

```
using System;
class Program
{
   [Obsolete]
   static void Foo() {}
   static void Main() {
#pragma warning disable 612
   Foo();
#pragma warning restore 612
   }
}
```

ДЖОЗЕФ АЛЬБАХАРИ

Компилятор генерирует предупреждение, когда он обнаруживает условие, которое может свидетельствовать об ошибке в коде. Из-за возможности «ложной тревоги» способность отключать отдельные предупреждения для выбранных строк кода является важной в смысле обеспечения хорошего соотношения сигнал/шум — так, чтобы действительные ошибки были замечены. Это существенно также в случае, когда вы хотите настроить компилятор таким образом, чтобы он обращался с предупреждениями как с ошибками — как это делаю в своих проектах я.

Глава 3

Основные понятия

3.1. Запуск приложения

Сборка, которая имеет **точку входа**, называется **приложением**. При запуске приложения создается новый **домен** (область) **приложения**. На одном компьютере могут одновременно существовать различные копии одного приложения, и каждая будет иметь собственный домен приложения.

Домен приложения обеспечивает изоляцию приложения, действуя как контейнер для состояния приложения. Домен приложения действует как контейнер и как граница для типов, определенных в приложении, и для используемых им библиотек классов. Типы, загруженные в один домен приложения, отличны от таких же типов, загруженных в другой домен приложения, и экземпляры объектов не могут совместно использоваться напрямую различными доменами приложения. Например, каждый домен приложения имеет свою копию статических переменных этих типов, и статический конструктор для типа запускается как минимум один раз для каждого домена. В различных реализациях могут применяться разные политики или механизмы создания и уничтожения доменов приложения.

Запуск приложения происходит, когда окружение вызывает метод, который является точкой входа приложения. Этот метод всегда именуется `Main` и имеет одну из следующих сигнатур:

```
static void Main() {...}
static void Main(string[] args) {... }
static int Main() {...}
static int Main(string [] args) {...}
```

Как видно из этих примеров, точка входа может возвращать (необязательно) тип `int`. Возвращаемое значение используется в процессе завершения работы приложения (раздел 3.2).

Метод, являющийся точкой входа, может иметь один необязательный формальный параметр. Параметр может иметь любое имя, но его тип должен быть `string[]`. Если формальный параметр присутствует, то окружение создает и передает аргумент `string[]`, содержащий аргументы командной строки, которые были заданы при запуске приложения. Аргумент `string[]` не может иметь значение `null`, но может иметь нулевую длину, если аргументы командной строки отсутствуют.

Так как C# поддерживает перегрузку методов, класс или структура могут содержать несколько определений одного и того же метода, при условии что каждый из них имеет отличную от других сигнатуру. Однако внутри одной программы никакой класс и никакая структура не могут содержать более одного метода, именуемого `Main`

и предназначенного для использования в качестве точки входа приложения. Другие перегруженные версии `Main` разрешены при условии, что они имеют более одного параметра или их единственный параметр имеет тип, отличный от `string[]`.

Приложение может состоять из множества классов или структур. Метод, именуемый `Main` и предназначенный для использования в качестве точки входа приложения, может содержаться более чем в одном классе или структуре. В таких случаях должен использоваться внешний механизм (например, параметр компилятора командной строки), чтобы выбрать один из этих методов `Main` в качестве точки входа.

> **ЭРИК ЛИППЕРТ**
>
> В компиляторе командной строки `csc` для этой цели есть ключ `/main:`.

В C# каждый метод должен быть определен как элемент класса или структуры. Обычно доступ к методу (раздел 3.5.1) определяется модификаторами доступа (раздел 10.3.5), указанными в объявлении, аналогичным образом доступ к типу определяется модификаторами доступа, указанными в его объявлении. Чтобы данный метод данного типа мог быть вызван, и тип и элемент должны быть доступны. Однако точка входа приложения — это особый случай: окружение может получить доступ к точке входа приложения независимо от указанного вида доступа к нему и к содержащим его типам.

Метод, являющийся точкой входа приложения, не может содержаться в объявлении обобщенного класса.

Во всех других отношениях точка входа в приложение ведет себя подобно методам, не являющихся точками входа.

3.2. Завершение работы приложения

При завершении работы приложения управление передается окружению.

Если тип возвращаемого значения метода *точки входа* приложения есть `int`, возвращаемое значение служит *кодом завершения приложения*. Назначением этого кода является передача информации окружению об успешном или неуспешном завершении.

Если тип возвращаемого значения метода точки входа приложения `void`, то достижение правой фигурной скобки `}`, завершающей метод, или выполнение оператора `return`, не содержащего выражения, дает в результате код завершения приложения `0`.

> **БИЛЛ ВАГНЕР**
>
> Следующее правило наглядно демонстрирует важное различие между C# и другой управляемой средой.

Перед завершением приложения вызываются деструкторы всех объектов, которые еще не были обработаны сборщиком мусора, если это не было запрещено (например, вызовом библиотечного метода `GC.SuppressFinalize`).

3.3. Объявления

Объявления в программе C# определяют составляющие программу элементы. Программы C# организуются с помощью пространств имен (глава 9), которые могут содержать объявления типов и вложенные объявления пространств имен. Объявления типов (раздел 9.6) используются для определения классов (глава 10), структур (раздел 10.14), интерфейсов (глава 13), перечислений (глава 14) и делегатов (глава 15). Виды элементов, разрешенных в объявлении типа, зависят от формы объявления. Например, объявление класса может содержать объявления констант (раздел 10.4), полей (раздел 10.5), методов (раздел 10.6), свойств (раздел 10.7), событий (раздел 10.8), индексаторов (раздел 10.9), операций (раздел 10.10), конструкторов экземпляров (раздел 10.11), статических конструкторов (раздел 10.12), деструкторов (раздел 10.13) и вложенных типов (раздел 10.3.8).

Объявление определяет имя в *области объявлений* (*declaration space)*, к которому оно принадлежит. Если два или более объявлений описывают элементы с одинаковыми именами в *одной области* объявлений, возникает ошибка компиляции — за исключением перегруженных элементов. Область объявлений не может содержать элементы различных видов с одним и тем же именем (например, одноименные поле и метод).

ЭРИК ЛИППЕРТ

«Области объявлений» часто путают с «областями видимости» (scope). Хотя эти понятия и связаны, они служат для различных целей. Область видимости имени элемента есть фрагмент текста программы, в рамках которого на это имя можно ссылаться без дополнительных уточнений. Область объявлений элемента, напротив, есть фрагмент, в котором никакие два элемента не могут иметь одно и то же имя. (*Почти* во всех случаях — например, методы могут иметь одинаковые имена, если у них различаются сигнатуры, а типы могут иметь одинаковые имена, если они различаются по количеству типов-аргументов.)

Есть несколько различных типов областей объявлений.

- Внутри всех исходных файлов программы *объявления-элементов-пространства-имен*, не включенные в *объявления-пространств-имен*, являются элементами единого совместного пространства, называемого *глобальным пространством имен*.
- Внутри всех исходных файлов программы *объявления-элементов-пространства-имен* внутри *объявлений-пространств-имен*, которые имеют одно и то же полностью определенное имя, являются элементами единой совместной области объявлений.

- Каждый класс, структура или интерфейс создают новую область объявлений. Имена вводятся в эту область при помощи *объявлений-элементов-класса*, *объявлений-элементов-структуры*, *объявлений-элементов-интерфейса* и *параметров-типов*. За исключением объявлений перегруженных конструкторов экземпляров и статических конструкторов, класс или структура не могут содержать объявления элементов с именем, совпадающим с именем класса или структуры. Класс, структура или интерфейс допускают объявления перегруженных методов и индексаторов. Более того, класс или структура допускают объявления перегруженных конструкторов экземпляров и операций. Например, класс, структура или интерфейс могут содержать множество объявлений методов с одним и тем же именем, при условии различия сигнатур в этих объявлениях (раздел 3.6). Заметим, что родительские классы не влияют на область объявлений класса, а родительские интерфейсы не влияют на область объявлений интерфейсов. Таким образом, производный класс или интерфейс имеет возможность объявлять элемент с тем же именем, как у наследуемого элемента. Можно сказать, что такой элемент *скрывает* наследуемый элемент.
- Каждое объявление делегата создает новую область объявлений. Имена вводятся в эту область объявлений с помощью формальных параметров (*обычных-параметров* и *параметров-массивов*) и с помощью *параметров-типов*.
- Каждое объявление перечисления создает новую область объявлений. Имена вводятся в эту область объявлений с помощью *объявлений-элементов-перечисления*.
- Каждое объявление метода, индексатора, операции, конструктора экземпляра или анонимной функции создает новую область объявлений, называемую **областью объявлений локальной переменной**. Имена вводятся в эту область с помощью формальных параметров (*обычных-параметров* и *параметров-массивов*) и с помощью *параметров-типов*. Тело функционального элемента класса или анонимной функции, если оно присутствует, считается вложенным в область объявлений локальной переменной. Если область объявлений локальной переменной и вложенная область объявлений локальной переменной содержат элементы с одним и тем же именем, выдается ошибка. Таким образом, внутри вложенной области объявлений невозможно объявить локальную переменную или константу с тем же именем, что и у локальной переменной или константы в пространстве объявлений, включающем в себя данное вложенное пространство. Две области объявлений могут содержать элементы с одинаковыми именами в том случае, если ни одна из них не содержит другую.
- Каждый *блок* или *switch-блок*, так же как каждый оператор *for*, *foreach* и *using*, создают область имен локальных переменных для локальных переменных и локальных констант. Имена вводятся в эту область объявлений с помощью *объявлений-локальных-переменных* и *объявлений-локальных-констант*. Заметим, что блоки внутри тела функционального элемента класса либо анонимной функции вложены в область объявлений локальных переменных, объявленную этими функциями для их параметров. Таким образом, будет ошибкой, например, метод, имеющий локальную переменную и параметр с одним и тем же именем.

- Каждый *блок* или *switch-блок* создает отдельную область объявлений для меток. Имена вводятся в это пространство имен с помощью *помеченных-операторов*, а ссылаются на эти имена в *операторах-перехода*. *Область объявлений меток* блока включает в себя любые вложенные блоки. Таким образом, внутри вложенного блока невозможно объявить метку с именем, совпадающим с меткой блока, содержащего в себе этот вложенный блок.

Текстуальный порядок, в котором объявляются имена, вообще говоря, не имеет значения. В частности, не имеет значения текстуальный порядок для объявления и использования пространств имен, констант, методов, свойств, событий, индексаторов, операций, конструкторов экземпляров, деструкторов, статических конструкторов и типов. Порядок объявлений имеет значение в следующих случаях:

- Порядок объявлений для полей и локальных переменных определяет порядок, в котором выполняются их инициализаторы (если таковые есть).
- Локальные переменные должны быть определены до того, как они будут использоваться (раздел 3.7).
- Порядок объявлений для элементов перечислений (раздел 14.3) имеет значение, когда опущены значения константных выражений.

Область объявлений пространств имен является «не замкнутой», и два объявления пространств имен с одинаковыми полностью определенными именами относятся к одной и той же области объявлений. Например:

```
namespace Megacorp.Data
{
   class Customer
   {
         ...
   }
}
namespace Megacorp.Data
{
   Class Order
   {
      ...
   }
}
```

Два приведенных объявления пространств имен относятся к одной и той же области объявлений, в данном случае объявляются два класса с полностью определенными именами: `Megacorp.Data.Customer` и `Megacorp.Data.Order`. Поскольку два объявления относятся к одному и тому же пространству имен, если каждое из них будет содержать объявление класса с одним и тем же именем, это приведет к ошибке компиляции.

> **БИЛЛ ВАГНЕР**
>
> Можно думать о пространствах имен как об инструменте для управления организацией кода на логическом уровне. Для сравнения, сборки управляют организацией кода на физическом уровне.

Как было сказано ранее, область объявлений блока включает в себя вложенные блоки. Таким образом, в следующем примере методы `F` и `G` дают ошибку компиляции, поскольку имя `i`, объявленное во внешнем блоке, не может быть снова объявлено во внутреннем блоке. Однако методы `H` и `I` работают, поскольку два `i` объявлены в отдельных не вложенных блоках.

```
class A
   {
      void F()
      {
         int i = 0;
         if (true)
         {
            int i = 1;
         }
      }
      void G()
      {
         if (true)
         {
            int i = 0;
         }
         int i = 1;
      }
      void H()
      {
         if (true)
         {
            int i = 0;
         }
         if (true)
         {
            int i = 1;
         }
      }
      void I()
      {
         for (int i = 0; i < 10; i++)
            H();
         for (int i = 0; i < 10; i++)
            H();
      }
   }
```

3.4. Элементы пространств имен и типов

Пространства имен и типы содержат **элементы**. В общем случае элементы некоторой сущности доступны через уточненное имя, которое начинается со ссылки на сущность, за которой после символа «.» следует имя элемента.

Элементы, входящие в некоторый тип, либо объявляются в объявлении типа, либо наследуются от родительского класса. Когда тип наследуется от родительского класса, все элементы родительского класса, за исключением конструкторов

экземпляров, деструкторов и статических конструкторов, становятся элементами производного типа. Объявленный вид доступа к элементам родительского класса не зависит от того, является ли элемент унаследованным — наследование распространяется на любые элементы, которые не являются конструкторами экземпляров, статическими конструкторами или деструкторами. Однако унаследованный элемент может оказаться недоступным в производном типе либо из-за вида доступа, который был объявлен (раздел 3.5.1), либо он может быть скрыт в соответствии с объявлением самого типа (раздел 3.7.1.2).

3.4.1. Элементы пространств имен

Пространства имен и типы, которые не содержатся в другом пространстве имен, являются элементами **глобального пространства имен**. Это означает, что их имена объявлены непосредственно в глобальном пространстве имен.

Пространства имен и типы, объявленные внутри пространства имен, являются элементами данного пространства. Это означает, что их имена объявлены непосредственно в области объявлений данного пространства имен.

Пространства имен не имеют ограничений доступа. Невозможно объявить пространства имен как `private`, `protected` или `internal`, их имена всегда находятся в открытом доступе.

3.4.2. Элементы структур

Элементами структуры являются элементы, объявленные в структуре, и элементы, унаследованные от непосредственного базового класса структур `System.ValueType` и корневого класса `object`.

Элементы простых типов прямо соответствуют элементам структурного типа, псевдонимами которых они являются:

- Элементы `sbyte` являются элементами структуры `System.SByte`.
- Элементы `byte` являются элементами структуры `System.Byte`.
- Элементы `short` являются элементами структуры `System.Int16`.
- Элементы `ushort` являются элементами структуры `System.UInt16`.
- Элементы `int` являются элементами структуры `System.Int32`.
- Элементы `uint` являются элементами структуры `System.UInt32`.
- Элементы `long` являются элементами структуры `System.Int64`.
- Элементы `ulong` являются элементами структуры `System.UInt64`.
- Элементы `char` являются элементами структуры `System.Char`.
- Элементы `float` являются элементами структуры `System.Single`.
- Элементы `double` являются элементами структуры `System.Double`.
- Элементы `decimal` являются элементами структуры `System.Decimal`.
- Элементы `bool` являются элементами структуры `System.Boolean`.

3.4.3. Элементы перечислений

Элементами перечисления являются константы, объявленные в перечислении, или элементы, унаследованные от непосредственного базового класса перечисления `System.Enum` и от непрямых родительских классов `System.ValueType` и `object`.

3.4.4. Элементы классов

Элементами класса являются элементы, объявленные в классе, и элементы, унаследованные от родительского класса (за исключением класса `object`, который не имеет родительского класса). Элементы, унаследованные от родительского класса, могут быть константами, полями, методами, свойствами, событиями, индексаторами, операциями и типами родительского класса, но не могут быть экземплярами конструкторов, деструкторами и статическими конструкторами родительского класса. Элементы родительского класса наследуются независимо от вида доступа к ним.

Объявление класса может содержать объявления констант, полей, методов, свойств, событий, индексаторов, операций, конструкторов экземпляров, деструкторов, статических конструкторов и типов.

Элементы типов `object` и `string` прямо соответствуют элементам типа класса, псевдонимами которых они являются:

- Элементы `object` являются элементами класса `System.Object`.
- Элементы `string` являются элементами класса `System.String`.

3.4.5. Элементы интерфейсов

Элементами интерфейса являются элементы, объявленные в интерфейсе и во всех базовых интерфейсах данного интерфейса. Элементы класса `object` не являются, строго говоря, элементами какого-либо интерфейса (раздел 13.2). Однако элементы класса `object` доступны при помощи поиска элементов (member lookup) в любом интерфейсном типе (раздел 7.4).

3.4.6. Элементы массивов

Элементами массива являются элементы, унаследованные от класса `System.Array`.

3.4.7. Элементы делегатов

Элементами делегатов являются элементы, унаследованные от класса `System.Delegate`.

ВЛАДИМИР РЕШЕТНИКОВ

В реализации Microsoft C# элементы делегатов также включают методы экземпляра `Invoke`, `BeginInvoke` и `EndInvoke` и элементы, унаследованные от класса `System.MulticastDelegate`.

ДЖОН СКИТ

Методы, о которых упоминал Владимир, `Invoke`, `BeginInvoke` и `EndInvoke`, не могут быть определены внутри типов `Delegate` или `MulticastDelegate`, так как они зависят от параметров и типа возвращаемого значения делегата. Это пример параметрического задания типов, когда обобщенные типы не могут помочь, даже если бы они существовали в первой версии C#.

3.5. Доступ к элементам

Объявления элементов позволяют управлять доступом к элементам. Вид доступа к элементу устанавливается с помощью объявления вида доступа (раздел 3.5.1) к элементу в сочетании с видом доступа к типу, непосредственно содержащему данный элемент, если он присутствует.

Когда доступ к отдельному элементу разрешен, можно сказать, что элемент *доступен*. Наоборот, когда доступ к отдельному элементу запрещен, можно сказать, что элемент *недоступен*. Доступ к элементу разрешен, когда место в тексте программы, где осуществляется доступ, входит в область доступа (раздел 3.5.2) элемента.

3.5.1. Объявление вида доступа

Объявление вида доступа к элементу определяется следующим образом:

- Доступ `public` реализуется при включении в объявление элемента модификатора `public`. Интуитивно понятное значение «public» есть «доступ не ограничен».
- Доступ `protected` реализуется при включении в объявление элемента модификатора `protected`. Интуитивно понятие «protected» воспринимается как «доступ ограничен содержащим элемент классом и типами, производными от этого класса».
- Доступ `internal` реализуется при включении в объявление элемента модификатора `internal`. Интуитивно понятие «internal» воспринимается как «доступ ограничен данной программой».
- Доступ `protected internal` (что означает `protected` *или* `internal`) реализуется при включении в объявление элемента обоих модификаторов `protected` и `internal`. Интуитивно понятие «protected internal» воспринимается как «доступ ограничен программой или типами, производными от содержащего элемент класса».

- Доступ `private` реализуется при включении в объявление элемента модификатора `private`. Интуитивно понятие «private» воспринимается как «доступ ограничен типом, содержащим элемент».

> **ДЖЕСС ЛИБЕРТИ**
>
> Несмотря на то что существует доступ по умолчанию, хорошим стилем программирования является явное объявление видов доступа. Это делает код легче для чтения и намного легче для понимания.

> **ДЖОН СКИТ**
>
> Доступ по умолчанию выбран в C# очень правильно: почти всегда доступен наиболее ограниченный уровень, за исключением частей свойства `getter/setter`, доступ к которым можно ограничить больше, чем это указано в общем объявлении свойства. Я обычно предпочитал оставлять неявно заданный вид доступа, но через какое-то время должен был согласиться с точкой зрения Джесса. То, что вы задаете что-либо явно, показывает, что вы осведомлены о существующем выборе и что вы сознательно выбрали определенный параметр. Если же вы делаете выбор неявно — то это может быть и оттого, что вы хотите оставить этот параметр, и оттого, что вы забыли, что нужно было сделать этот выбор.
>
> В зависимости от контекста, в котором существует объявление элемента, разрешено объявлять только определенные виды доступа. Когда объявление элемента не содержит никаких модификаторов доступа, доступ по умолчанию определяется контекстом объявления.

- Пространства имен имеют доступ по умолчанию `public`. В объявлении пространства имен не разрешены никакие модификаторы доступа.
- Типы, объявленные в единицах трансляции или в пространствах имен, можно объявлять как `public` или `internal`; доступ по умолчанию — `internal`.
- Элементы класса можно объявлять с любым из пяти видов доступа; доступ по умолчанию — `private`. (Заметим, что тип, объявленный как элемент класса, можно объявлять с любым из пяти видов доступа, а тип, объявленный как элемент пространства имен, может иметь только доступ `public` или `internal`.)

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Если бесплодный класс объявляется как элемент с модификаторами `protected` или `protected internal`, выдается предупреждение. Если статический класс объявляется как элемент с модификаторами `protected` или `protected internal`, выдается ошибка компиляции (CS1057).

- Структурные элементы можно объявлять как `public`, `internal` или `private`; доступ по умолчанию — `private`, поскольку от структур нельзя наследовать. Элементы, объявленные в структуре (то есть не унаследованные этой струк-

турой), нельзя объявлять как `protected` или `protected internal`. (Заметим, что тип, объявленный как элемент структуры, можно объявлять как `public`, `internal` или `private`, а тип, объявленный как элемент пространства имен, может иметь только доступ `public` или `internal`.)

- Интерфейсные элементы имеют доступ по умолчанию `public`. В объявлениях элементов интерфейса не разрешены никакие модификаторы доступа.
- Элементы перечислений имеют доступ по умолчанию `public`. В объявлениях элементов перечислений не разрешены никакие модификаторы доступа.

ВЛАДИМИР РЕШЕТНИКОВ

Выражение «элементы перечислений» здесь означает «элементы, объявленные в перечислении». Перечисление также наследует элементы от своих родительских классов `System.Enum`, `System.ValueType` и `System.Object`, и эти элементы могут не быть `public`.

ДЖОЗЕФ АЛЬБАХАРИ

Рациональное зерно этих правил в том, что доступ по умолчанию для любых конструкций есть минимальный доступ, который требуется для того, чтобы их можно было использовать. Минимизация доступа является положительным фактором для поддержки инкапсуляции.

ДЖЕСС ЛИБЕРТИ

Как я уже говорил, полезной практикой является задавать вид доступа явно; это делает код легче для понимания.

ЭРИК ЛИППЕРТ

Есть различие между «объявленным» доступом и действительно использующимся. Например, метод, объявленный как `public` в классе, объявленном как `internal`, является для большинства практических целей внутренним методом.

Хороший способ понять этот момент состоит в том, чтобы осознать, что элемент класса `public` является `public` только для объектов, которые имеют доступ к этому классу.

ДЖОН СКИТ

Одно важное исключение из ремарки Эрика возникает в том случае, когда вы переопределяете `public` метод внутри `internal` (или даже `private`) класса — включая реализацию интерфейса. Реализации интерфейса часто могут встречаться внутри `internal` классов, но они могут оставаться доступными для других сборок через интерфейс.

3.5.2. Области доступности

Область доступности (accessibility domain) элемента включает в себя разделы программного текста (возможно, разрозненные), в которых доступ к элементу разрешен. Для описания области доступности элемента введем некоторые определения: будем называть его элементом **верхнего уровня**, если он не объявлен внутри типа, и будем называть элемент **вложенным**, если он объявлен внутри другого типа. Далее, **программный текст** программы определяется как весь программный текст, содержащийся во всех исходных файлах программы, а *программный текст типа* определяется как весь программный текст, содержащийся в *объявлениях-типов* этого типа (включая, возможно, типы, вложенные в данный тип).

Область доступности предопределенного типа (например, `object`, `int` или `double`) не ограничена.

Область доступности неограниченного типа верхнего уровня `T` (раздел 4.4.3), объявленного в программе `P`, определяется следующим образом:

- Если объявленный вид доступа к `T` есть `public`, то областью доступности `T` является программный текст `P` или любой программы, которая ссылается на `P`.
- Если объявленный вид доступа к `T` есть `internal`, то областью доступности `T` является программный текст `P`.

Из этих определений следует, что областью доступности неограниченного типа верхнего уровня является по меньшей мере программный текст программы, в которой этот тип был объявлен.

Область доступности сконструированного типа $T<A_1, ..., A_N>$ есть пересечение области доступности неограниченного обобщенного типа T и областей доступности аргументов-типов $A_1, ..., A_N$.

Область доступности вложенного элемента `M`, объявленного в типе `T` в программе `P`, определяется следующим образом (отметим, что `M` также может быть типом):

- Если `M` объявлен как `public`, областью доступности `M` является область доступности `T`.
- Если `M` объявлен как `protected internal`: обозначим через `D` объединение программного текста `P` и программного текста любых типов, производных от `T`, которые объявлены вне `P`; тогда областью доступности `M` будет пересечение областей доступности `T` и `D`.
- Если `M` объявлен как `protected`: обозначим через `D` объединение программного текста `P` и программного текста любых типов, производных от `T`; тогда областью доступности `M` будет пересечение областей доступности `T` и `D`.
- Если `M` объявлен как `internal`, областью доступности `M` является пересечение области доступности `T` с программным текстом `P`.
- Если `M` объявлен как `private`, областью доступности `M` является программный текст `T`.

Из этих определений следует, что областью доступности вложенного элемента является по меньшей мере программный текст типа, в котором элемент был объявлен.

Далее, отсюда следует, что область доступности элемента никогда не может включать в себя больше, чем область доступности типа, в котором элемент был объявлен.

Говоря неформальным языком, при доступе к типу или элементу `M` для проверки, разрешен ли доступ, выполняются следующие шаги:

- Если `M` объявлен внутри типа (в противоположность случаю, когда он объявлен в пространстве имен или единице компиляции), то если этот тип недоступен, выдается ошибка компиляции.
- Далее, если `M` объявлен как `public`, доступ разрешен.
- Иначе, если `M` объявлен как `protected internal`, доступ разрешен, если он выполняется в программе, в которой объявлен `M`, или в классе, производном от того, в котором объявлен `M`, и выполняется через тип производного класса (раздел 3.5.3).
- Иначе, если `M` объявлен как `protected`, доступ разрешен, если он выполняется в классе, в котором объявлен `M`, или в классе, производном от класса, в котором объявлен `M`, и выполняется через тип производного класса (раздел 3.5.3).
- Иначе, если `M` объявлен как `internal`, доступ разрешен, если он выполняется внутри программы, в которой объявлен `M`.
- Иначе, если `M` объявлен как `private`, доступ разрешен, если он выполняется внутри типа, в котором объявлен `M`.
- Иначе тип или элемент недоступен, и выдается ошибка компиляции.

В примере

```
public class A
{
   public static int X;
   internal static int Y;
   private static int Z;
}

internal class B
{
   public static int X;
   internal static int Y;
   private static int Z;

   public class C
   {
      public static int X;
      internal static int Y;
      private static int Z;
   }

   private class D
   {
      public static int X;
      internal static int Y;
      private static int Z;
   }
}
```

классы и элементы имеют следующие области доступности:

- Области доступности `A` и `A.X` не ограничены.
- Областью доступности `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X` и `B.C.Y` является программный текст содержащей их программы.
- Областью доступности `A.Z` является программный текст `A`.
- Областью доступности `B.Z` и `B.D` является программный текст `B`, включая программный текст `B.C` и `B.D`.
- Областью доступности `B.C.Z` является программный текст `B.C`.
- Областью доступности `B.D.X` и `B.D.Y` является программный текст `B`, включая программный текст `B.C` и `B.D`.
- Областью доступности `B.D.Z` является программный текст `B.D`.

Как показывает этот пример, область доступности элемента никогда не может быть больше, чем область доступности содержащего его типа. Например, хотя все элементы `X` объявлены как `public`, все они, кроме `A.X`, имеют область доступности, ограниченную содержащим их типом.

ДЖОЗЕФ АЛЬБАХАРИ

Объявление элемента `public` внутри типа `internal` может показаться неэффективным, учитывая, что видимость элемента будет ограничена `internal`. Оно, однако, может иметь смысл, если значение модификатора `public` интерпретировать как «имеющий такую же видимость, как содержащий его тип».
Можно задать хороший вопрос, когда вы принимаете решение, объявлять ли элемент, объявленный в `internal`типе, как `public` или как `internal`: если тип позже станет `public`, хочу ли я, чтобы элемент тоже стал `public`? Если ответ «да», то можно согласиться с тем, чтобы объявить его как `public` с самого начала.

ДЖЕСС ЛИБЕРТИ

Хотя можно привести разумные примеры для каждого из случаев, упоминавшихся выше, в практике программирования предпочтительнее использовать менее сложные и более очевидные виды доступа, чтобы уменьшить количество ошибок и сделать код легче для понимания. Я написал сотни приложений, вполне жизнеспособных в коммерческом отношении, не используя ничего кроме `public`, `private` и `protected`.

Как было описано в разделе 3.4, все элементы родительского класса, за исключением конструкторов экземпляров, деструкторов и статических конструкторов, наследуются производными типами. Это относится даже к элементам `private` родительского класса. Однако область доступности элементов `private` включает в себя только программный текст типа, в котором объявлен элемент. В примере

```
class A
{
   int x;
      static void F(B b)
   {
```

```
      b.x = 1;            // Правильно
   }
}
class B : A
{
   static void F(B b)
   {
      b.x = 1;            // Ошибка: нет доступа к x
   }
}
```

класс `B` наследует элемент `x` с доступом `private` от класса `A`. Поскольку элемент имеет доступ `private`, он доступен только внутри *тела-класса* A. Таким образом, доступ к `b.x` успешно выполняется в методе `A.F`, но не допускается в методе `B.F`.

> **БИЛЛ ВАГНЕР**
> Заметьте, что недоступные методы также избавляют от необходимости вводить модификатор `new` для `B:F()`.

3.5.3. Доступ protected для элементов экземпляров

Если `protected` элемент экземпляра доступен за пределами программного текста класса, в котором он объявлен, и если `protected internal` элемент экземпляра доступен вне программного текста программы, в которой он объявлен, доступ должен происходить внутри объявления класса, производного от класса, в котором элемент был объявлен. Более того, требуется, чтобы доступ осуществлялся через экземпляр этого производного типа класса или типа класса, сконструированного на его основе. Это ограничение предотвращает возможность доступа одного производного класса к защищенным элементам других производных классов, даже когда элементы унаследованы от одного и того же родительского класса.

> **ЭРИК ЛИППЕРТ**
> Например, предположим, что у вас есть родительский класс `Animal` и два производных класса `Mammal` и `Reptile` и `Animal` имеет защищенный метод `Feed()`. Тогда код `Mammal` может вызвать метод `Feed()` для `Mammal` или для любого подкласса `Mammal` (скажем, `Tiger`).
> Код `Mammal` не может вызвать `Feed()` для выражения типа `Reptile`, поскольку между `Mammal` и `Reptile` нет отношения наследования.
> Более того, поскольку выражение типа `Animal` может иметь значение `Reptile` в какой-то момент выполнения программы, код `Mammal` не может вызвать `Feed()` для выражения типа `Animal`.

Пусть `B` — родительский класс, в котором объявлен `protected` элемент `M`, и пусть `D` — дочерний класс `B`. Внутри *тела-класса* `D` доступ к `M` может принимать одну из следующих форм:

- Неуточненное *имя-типа* или *первичное-выражение* в форме `M`.
- *Первичное-выражение* в форме `E.M`, при условии, что тип `E` есть `T` или тип класса, производного от `T`, где `T` есть тип класса `D`, или тип класса, сконструированного на основе `D`.
- *Первичное-выражение* в форме `base.M`.

В дополнение к этим формам доступа производный класс может иметь доступ к `protected` конструктору экземпляра родительского класса в *инициализаторе-конструктора* (раздел 10.11.1).

ВЛАДИМИР РЕШЕТНИКОВ

Просто считайте, что другие формы доступа не разрешены. Например, производный класс не может вызвать `protected` конструктор своего родительского класса в операции `new`:

```
class Base
{
   protected Base() { }
}
class Derived : Base
{
   static void Main()
   {
      new Base();             // Error CS0122: 'Base.Base()' is
                              // inaccessible due to its protection level
                              // (Ошибка CS0122: 'Base.Base()'
                              // недостижим из-за уровня защиты)
   }
}
```

В примере

```
public class A
{
   protected int x;
   static void F(A a, B b)
   {
      a.x = 1;              // Правильно
      b.x = 1;              // Правильно
   }
}
public class B : A
{
   static void F(A a, B b)
   {
      a.x = 1;              // Error: must access through instance of B
                            // (Ошибка: требуется доступ через экземпляр B)
      b.x = 1;              // Правильно
   }
}
```

внутри `A` возможен доступ к `x` через экземпляры как класса `A`, так и класса `B`, так как в любом случае доступ осуществляется через экземпляры класса `A` или класса,

производного от A. Однако внутри B невозможен доступ к x через экземпляры класса A, так как A не является потомком B.

В примере

```
class C<T>
{
   protected T x;
}
class D<T> : C<T>
{
   static void F()
   {
      D<T> dt = new D<T>();
      D<int> di = new D<int>();
      D<string> ds = new D<string>();
      dt.x = default(T);
      di.x = 123;
      ds.x = "test";
   }
}
```

три присваивания для x разрешены, поскольку все они используют экземпляры классов, сконструированных на основе обобщенного типа.

> **КРИС СЕЛЛЗ**
>
> Чтобы минимизировать площадь поверхности класса или пространства имен, я рекомендую сохранять статус `private/internal` до тех пор, пока не будет необходимости в более широкой области видимости. Рефакторинг в данном случае ваш друг.

> **БИЛЛ ВАГНЕР**
>
> Эти правила существуют, чтобы в разных сборках компоненты можно было изменять раздельно. Не обязательно включать эти правила в ваши обычные паттерны проектирования.

3.5.4. Ограничения доступа

Некоторые конструкции в языке C# требуют, чтобы тип был **по меньшей мере так же доступен**, как элемент или другой тип. Говорят, что тип T так же доступен, как элемент или тип M, если область доступности T есть надмножество области доступности M. Другими словами, T по меньшей мере так же доступен, как M, если T доступен во всех контекстах, в которых доступен M.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Для целей данного раздела будут рассматриваться только модификаторы доступа. Например, если `protected` элемент объявлен в `public sealed` классе, тот факт, что этот класс не может иметь потомков (и, следовательно, область доступа этого элемента не может включать каких-либо потомков), не рассматривается.

Существуют следующие ограничения доступа:

- Прямой предок класса должен быть по меньшей мере так же доступен, как сам класс.
- Явные интерфейсы-предки интерфейсного типа должны быть по меньшей мере так же доступны, как он сам.
- Тип результата и параметры-типы делегата должны быть по меньшей мере так же доступны, как он сам.
- Тип константы должен быть по меньшей мере так же доступен, как она сама.
- Тип поля должен быть по меньшей мере так же доступен, как оно само.
- Тип свойства должен быть по меньшей мере так же доступен, как оно само.
- Тип события должен быть по меньшей мере так же доступен, как оно само.
- Тип результата и параметры-типы индексатора должны быть по меньшей мере так же доступны, как сам индексатор.
- Тип результата и параметры-типы операции должны быть по меньшей мере так же доступны, как сама операция.
- Параметры-типы конструктора экземпляра должны быть по меньшей мере так же доступны, как сам конструктор экземпляра.

В примере

```
class A {...}
public class B: A {...}
```

объявление класса `B` приведет к ошибке компиляции, поскольку `A` менее доступен, чем `B`.

Аналогично, в примере

```
class A {...}
public class B
{
   A F() {...}
   internal A G() {...}
   public A H() {...}
}
```

метод `H` в классе `B` приведет к ошибке компиляции, поскольку тип результата `A` менее доступен, чем метод.

3.6. Сигнатуры и перегрузка

Методы, конструкторы экземпляра, индексаторы и операции характеризуются своими **сигнатурами**:

- Сигнатура метода включает в себя имя метода, количество параметров-типов, а также тип и вид (параметр-значение, параметр-ссылка или выходной параметр) каждого из его формальных параметров, рассматриваемых по порядку слева направо. При таком рассмотрении любой параметр-тип метода, который встречается в виде формального параметра, идентифицируется не по имени, а по его положению в списке аргументов-типов метода. Сигнатура метода не включа-

ет в себя тип результата, модификатор `params`, который может быть определен для самого правого параметра, и возможные ограничения параметра-типа.

> **ДЖОН СКИТ**
>
> Использование при перегрузке числа параметров-типов интересно — и существует также на уровне типа (рассмотрим, например, типы .NET `System.Nullable` и `System.Nullable<T>`). Тот факт, что ограничения не принимаются во внимание, иногда вызывает раздражение. В некоторых случаях это может быть полезно, например, чтобы написать следующее:
>
> ```
> void Process<T>(T reference) where T : class { ... }
> void Process<T>(T value) where T : struct { ... }
> ```
>
> но может создавать дополнительные сложности в области, которая и так весьма коварна. Известно, что мозги разработчиков начинают плавиться, когда они встречаются с перегрузкой и выведением типов (type inference).

- Сигнатура конструктора экземпляра включает в себя тип и вид (параметр-значение, параметр-ссылка или выходной параметр) каждого из его формальных параметров, рассматриваемых по порядку слева направо. Сигнатура конструктора экземпляра не включает в себя модификатор `params`, который может быть определен для самого правого параметра.
- Сигнатура индексатора включает в себя тип каждого из его формальных параметров, рассматриваемых по порядку слева направо. Сигнатура индексатора не включает в себя тип элемента, а также модификатор `params`, который может быть определен для самого правого параметра.
- Сигнатура операции включает в себя имя операции и тип каждого из ее формальных параметров, рассматриваемых по порядку слева направо. Сигнатура операции не включает в себя тип результата.

Сигнатуры делают возможным механизм *перегрузки* элементов классов, структур и интерфейсов:

- Перегрузка методов позволяет классу, структуре или интерфейсу объявлять несколько методов с одним и тем же именем при условии уникальности их сигнатур внутри класса, структуры или интерфейса.
- Перегрузка конструкторов экземпляров позволяет классу или структуре объявлять несколько конструкторов экземпляров при условии уникальности их сигнатур внутри класса или структуры.
- Перегрузка индексаторов позволяет классу, структуре или интерфейсу объявлять несколько индексаторов при условии уникальности их сигнатур внутри класса, структуры или интерфейса.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Есть одно исключение для требования уникальности для сигнатур индексаторов: если индексатор является явной реализацией интерфейса, его сигнатуры проверяются на уникальность только внутри явно реализованных индексаторов:
>
> *продолжение* ⇨

```
class A : IIndexable
{
   int IIndexable.this[int x]
   {
      get { /* ... */ }
   }
   public int this[int x] // Okay
   {
      get { /* ... */ }
   }
}
```

- Перегрузка операций позволяет классу или структуре объявлять множество операций с одним и тем же именем при условии уникальности их сигнатур внутри класса или структуры.

Хотя модификаторы параметров `out` и `ref` рассматриваются как часть сигнатуры, элементы, объявленные в одном типе, не могут различаться в сигнатуре только за счет `ref` и `out`. Если два элемента будут объявлены в одном и том же типе с сигнатурами, у которых все параметры одинаковы в обоих методах, только модификатор `out` заменен на `ref`, будет выдана ошибка компиляции. В других случаях сравнения сигнатур (то есть в случаях скрытия или переопределения) `ref` и `out` рассматриваются как значимая часть сигнатуры и не могут заменять друг друга. (Это ограничение легко позволяет транслировать программы на C# для работы в общеязыковой инфраструктуре (CLI), в которой не существует способа определить методы, различающиеся только `ref` и `out`.)

Что касается сигнатур, типы `object` и `dynamic` рассматриваются как одинаковые. Элементы, объявленные в одном типе, не могут, таким образом, различаться в сигнатурах только за счет `object` и `dynamic`.

Следующий пример показывает ряд объявлений перегруженного метода с их сигнатурами.

```
interface ITest
   {
      void F();                // F()
      void F(int x);           // F(int)
      void F(ref int x);       // F(ref int)
      void F(out int x);       // F(out int)    ошибка
      void F(int x, int y);    // F(int, int)
      int F(string s);         // F(string)
      int F(int x);            // F(int) ошибка
      void F(string[] a);      // F(string[])
      void F(params string[] a);  // F(string[]) ошибка
   }
```

Отметим, что любой из модификаторов параметров `ref` и `out` (раздел 10.6.1) является частью сигнатуры. Таким образом, `F(int)` и `F(ref int)` являются уникальными сигнатурами. Однако `F(ref int)` и `F(out int)` не могут быть объявлены внутри одного и того же интерфейса, поскольку их сигнатуры различаются только за счет `ref` и `out`. Также отметим, что тип результата и модификатор `params` не являются частью сигнатуры, так что при перегрузке невозможно основываться

только на типе результата или на наличии или отсутствии модификатора `params`. Таким образом, объявления методов `F(int)` и `F(params string[])`, приведенные выше, приводят к ошибке компиляции.

3.7. Области видимости

Область видимости (scope) есть фрагмент текста программы, внутри которого можно обращаться к именованной сущности без уточнения имени. Области видимости могут быть **вложенными**, и во внутренней области может быть переопределено значение имени из внешней области (это, однако, не отменяет ограничение, описанное в разделе 3.3, касающееся того, что во вложенном блоке нельзя объявлять локальную переменную с тем же самым именем, что в окружающем его блоке). В этом случае имя из внешней области видимости называется **скрытым** во фрагменте текста программы, относящегося к внутренней области видимости, и доступ ко внешнему имени возможен только с помощью уточнения имени.

- Областью видимости элемента пространства имен, объявленного в *объявлении-элемента-пространства-имен* (раздел 9.5), не содержащегося в другом *объявлении-пространства-имен*, является весь текст программы.
- Областью видимости элемента пространства имен, объявленного в *объявлении-элемента-пространства-имен* внутри *объявления-пространства-имен* с полностью уточненным именем `N`, является *тело-пространства-имен* любого *объявления-пространства-имен*, полностью уточненное имя которого либо `N`, либо начинается с N с точкой.
- Область действия имени, определяемая *директивой-внешнего-псевдонима* (*extern-alias-directive*), распространяется на *директивы-using*, *глобальные-атрибуты* и *объявления-элементов-пространства-имен единицы-компиляции* или *тела-пространства-имен*, непосредственно содержащих эти элементы. *Директива-внешнего-псевдонима* не вносит новых элементов в нижележащую область объявлений. Иными словами, *директива-внешнего-псевдонима* не транзитивна, она действует только на единицу компиляции или тело пространства имен, в которых она встречается.
- Область действия имени, определенная или импортированная *директивой-using* (раздел 9.4), распространяется на *объявления-элементов–пространства-имен единицы-компиляции* или *тела-пространства-имен*, в которых появляется эта директива. *Директива-using* может делать доступными ноль или более пространств имен или имен типов внутри определенной *единицы-компиляции* или *тела-пространства-имен*, но не вносит новых элементов в нижележащую область объявлений. Иными словами, *директива-using* не транзитивна, она действует только на *единицу-компиляции* или на *тело-пространства-имен*, в которых она встречается.
- Областью видимости параметра-типа, объявленного в *списке-параметров-типов* в *объявлении-класса* (раздел 10.1), является *базовый-класс*, *ограничения-параметров-типов* и *тело-класса* в данном *объявлении-класса*.

- Областью видимости параметра-типа, объявленного в *списке-параметров-типов* в *объявлении-структуры* (раздел 11.1), являются *интерфейсы-структуры*, *ограничения-параметров-типа* и *тело-структуры* в данном *объявлении-структуры*.
- Областью видимости параметра-типа, объявленного в *списке-параметров-типов* в *объявлении-интерфейса* (раздел 13.1), является *базовый-интерфейс*, *ограничения-параметров-типа* и *тело-интерфейса* в данном *объявлении-интерфейса*.
- Областью видимости параметра-типа, объявленного в *списке-параметров-типов* в *объявлении-делегата* (раздел 15.1), является *тип-результата*, *список-формальных-параметров* и *ограничения-параметров-типа* в данном *объявлении-делегата*.
- Областью видимости элемента, объявленного в *объявлении-элемента-класса* (раздел 10.1.6), является *тело-класса*, в котором встречается это объявление. Кроме того, область видимости элемента класса распространяется на *тело-класса* производных от него классов, которые входят в область доступности элемента (раздел 3.5.2).
- Областью видимости элемента, объявленного в *объявлении-элемента-структуры* (раздел 11.2), является *тело-структуры*, в которой встречается это объявление.
- Областью видимости элемента, объявленного в *объявлении-элемента-перечисления* (раздел 14.3), является *тело-перечисления*, в котором встречается это объявление.
- Областью видимости параметра, объявленного в *объявлении-метода* (раздел 10.6), является *тело-метода* в этом *объявлении-метода*.
- Областью видимости параметра, объявленного в *объявлении-индексатора* (раздел 10.9), является *объявление-кодов-доступа* в этом *объявлении-индексатора*.
- Областью видимости параметра, объявленного в *объявлении-операции* (раздел 10.10), является *блок* в этом *объявлении-операции*.
- Областью видимости параметра, объявленного в *объявлении-конструктора* (раздел 10.11), является *инициализатор-конструктора* и *блок* в этом объявлении конструктора.
- Областью видимости параметра, объявленного в *лямбда-выражении* (раздел 7.15), является *тело-лямбда-выражения* этого *лямбда-выражения*.
- Областью видимости параметра, объявленного в *выражении-анонимного-метода* (раздел 7.15), является *блок* этого *выражения-анонимного-метода*.
- Областью видимости метки, объявленной в *помеченном-операторе* (раздел 8.4), является *блок*, в котором встречается это объявление.
- Областью видимости локальной переменной, объявленной в *объявлении-локальной-переменной* (раздел 8.5.1), является *блок*, в котором встречается это объявление.

- Областью видимости локальной переменной, объявленной в *блоке-switch* оператора `switch` (раздел 8.7.2), является этот *блок-switch*.
- Областью видимости локальной переменной, объявленной в *инициализаторе-for* оператора цикла `for` (раздел 8.8.3), являются *инициализатор-for*, *условие-for*, *итератор-for* и *оператор*, содержащийся в этом операторе `for`.
- Областью видимости локальной константы, объявленной в *объявлении-локальной-константы* (раздел 8.5.2), является *блок*, в котором встречается это объявление. Ссылка на локальную константу в позиции текста, предшествующей *описателю-константы* этой константы, приведет к ошибке компиляции.
- Область видимости переменной, объявленной как часть *оператора-foreach*, *оператора-using*, *оператора-lock* или *выражения-запроса*, определяется областью распространения данной конструкции.

> **ДЖЕСС ЛИБЕРТИ**
>
> Этот список является классическим, весьма наглядным примером различия между тем, что возможно, и тем, что желательно. Области видимости, так же как имена методов и вообще все в вашей программе, должны быть прозрачными и недвусмысленными, насколько это возможно, — но не более того!

Внутри области видимости элемента пространства имен, класса, структуры или перечисления можно ссылаться на элемент в позиции текста, предшествующей объявлению этого элемента. В примере

```
class A
{
   void F()
   {
      i = 1;
   }
   int i = 0;
}
```

`F` законно ссылается на `i`, до того, как она объявлена.

Внутри области видимости локальной переменной при ссылке на локальную переменную в позиции текста, предшествующей *описателю-локальной-переменной* этой локальной переменной, возникает ошибка компиляции. Например:

```
class A
{
   int i = 0;
   void F()
   {
      i = 1;                  // Ошибка: использование раньше описания
      int i;
      i = 2;
   }
   void G()
   {
        int j = (j = 1);      // Правильно
   }
```

продолжение ☞

```
   void H()
   {
        int a = 1, b = ++a;    // Правильно
   }
}
```

В методе F первое присваивание переменной i не ссылается на поле, объявленное во внешней области видимости. Напротив, ссылка на локальную переменную приводит к ошибке компиляции, поскольку она текстуально предшествует объявлению переменной. В методе G использование j в инициализаторе для объявления j корректно, поскольку оно не предшествует *описателю-локальной-переменной*. В методе H второй *описатель-локальной-переменной* корректно ссылается на локальную переменную, объявленную в предшествующем *описателе-локальной-переменной* внутри того же самого *объявления-локальной-переменной*.

Правила для областей видимости локальных переменных созданы так, чтобы гарантировать, что значение имени, используемого в контексте выражения, всегда одно и то же в пределах блока. Если бы область видимости локальной переменной распространялась только от ее объявления до конца блока, то в приведенном выше примере первое присваивание было бы сделано переменной экземпляра, а второе — локальной переменной, что, возможно, привело бы к ошибке компиляции, если впоследствии операторы блока были бы переставлены.

Значение имени внутри блока может различаться в зависимости от контекста, в котором имя используется. В примере

```
using System;
class A { }
class Test
{
   static void Main()
   {
      string A = "hello, world";
      string s = A;                // Контекст выражения

      Type t = typeof(A);          // Контекст типа
      Console.WriteLine(s);        // Выводится "hello, world"
      Console.WriteLine(t);         // Выводится "A"
   }
}
```

имя A используется в контексте выражения для ссылки на локальную переменную A и в контексте типа для ссылки на класс A.

3.7.1. Скрытие имен

Область видимости объекта обычно распространяется на больший фрагмент программного текста, чем пространство объявления объекта. В частности, область видимости некоторой сущности может включать объявления, в которых вводятся новые области объявлений, содержащие сущности с тем же именем. Такие объявления делают первоначальную сущность **скрытой**. Напротив, сущность является **видимой**, если она не скрыта.

Скрытие имен происходит, когда области видимости частично перекрываются из-за вложения или наследования. Характеристики этих двух типов скрытия описаны в следующих разделах.

> **ДЖЕСС ЛИБЕРТИ**
> Я склоняюсь к мнению, что скрытие имен всегда можно рассматривать как ошибку, которая затрудняет понимание кода и которой всегда можно избежать.

3.7.1.1. Скрытие в результате вложения

Скрытие имен может происходить как результат вложения пространств имен или типов в пространства имен, как результат вложения типов в классы или структуры и как результат объявлений параметров и локальных переменных.

В примере

```
class A
{
   int i = 0;

   void F()
   {
      int i = 1;
   }

   void G()
   {
      i = 1;
   }
}
```

в методе `F` переменная экземпляра `i` скрыта локальной переменной `i`, но в методе `G` на `i` все еще ссылаются как на переменную экземпляра.

Когда имя во внутренней области видимости скрывает имя из внешней области видимости, оно скрывает все перегруженные значения этого имени. В примере

```
class Outer
{
   static void F(int i) { }
   static void F(string s) { }
   class Inner
   {
      void G()
      {
         F(1);                  // Вызывает Outer.Inner.F
         F("Hello");            // Ошибка
      }
      static void F(long l) { }
   }
}
```

`F(1)` вызывает метод `F`, объявленный в `Inner`, поскольку все внешние описания `F` скрыты объявлением во внутренней области. По той же самой причине вызов `F("Hello")` приводит к появлению ошибки компиляции.

ВЛАДИМИР РЕШЕТНИКОВ

Если вложенная область видимости содержит элемент с таким же именем, как элемент во внешней области видимости, элемент из внешней области видимости не всегда скрыт благодаря следующему правилу: если элемент *можно вызвать*, все *невызываемые элементы* (*non-invocable*) удаляются из набора (см. раздел 7.3).

```
class A
{
   static void Foo() { }
   class B
   {
      const int Foo = 1;
      void Bar()
      {
         Foo();                   // Правильно
      }
   }
}
```

3.7.1.2. Скрытие в результате наследования

Скрытие имен через наследование появляется, когда в классах или структурах переопределяют имена, унаследованные от родительских классов. Этот тип скрытия имени существует в одной из следующих форм:

- Константа, поле, свойство, событие или тип, описанные в классе или структуре, скрывают все элементы родительского класса с таким же именем.
- Метод, описанный в классе или структуре, скрывает все элементы родительского класса с таким же именем, не являющиеся методами, и все методы родительского класса с такими же сигнатурами (имя метода, количество параметров, модификаторы и типы).
- Индексатор, описанный в классе или структуре, скрывает все индексаторы родительского класса с такими же сигнатурами (количество параметров и типы).

Правила, определяющие объявления операций (раздел 10.10), делают невозможным объявление в производном классе операции с такой же сигнатурой, как у операции в родительском классе. Таким образом, операции никогда не скрывают друг друга.

В противоположность скрытию имени из внешней области видимости, скрытие доступного имени из унаследованной области видимости приводит к появлению предупреждения. В примере

```
class Base
{
   public void F() { }
}
class Derived : Base
{
   public void F() { }          // Предупреждение: скрытие унаследованного имени
}
```

объявление F в Derived приводит к выдаче предупреждения. Скрытие унаследованного имени, вообще говоря, не является ошибкой, но впоследствии может препятствовать раздельному изменению родительских классов. Например, ситуация в приведенном выше примере может привести к чему-то подобному, поскольку в более поздней версии Base описан метод F, которого не было в более ранней версии класса. Такая ситуация может привести к ошибке, поскольку *любое* изменение родительского класса в отдельной версии библиотеки классов может послужить причиной того, что производные классы окажутся неверными.

Предупреждение, вызванное скрытием унаследованного имени, может быть устранено при помощи модификатора new:

```
class Base
{
   public void F() { }
}
class Derived : Base
{
   new public void F() { }
}
```

Модификатор new показывает, что F в Derived является «новым» и что действительно существовало намерение скрыть унаследованный элемент.

Объявление нового элемента скрывает унаследованный элемент только внутри области видимости нового элемента.

```
class Base
{
   public static void F() { }
}
class Derived : Base
{
   new private static void F() { }   // Скрывает Base.F только в Derived
}
class MoreDerived : Derived
{
   static void G() { F(); }          // Вызывается Base.F
}
```

В этом примере объявление F в Derived скрывает F, который был унаследован от Base, но поскольку новый F в Derived имеет доступ private, его область видимости не должна распространяться на MoreDerived. Таким образом, вызов F() в MoreDerived.G допустим и вызывает метод Base.F.

КРИС СЕЛЛЗ

Если вы используете new, чтобы скрыть метод экземпляра базового класса, вы почти всегда будете разочарованы, если только это не делается для того, чтобы вызывающая сторона могла просто выполнить преобразование типа к родительскому классу, чтобы получить «скрытый» метод. Например:

```
class Base { public void F() {} }
class Derived : Base { new public void F() {} }
Derived d = new Derived();
((Base)d).F();       // Base.F не настолько скрыт, как вы бы этого хотели
```

Будет намного лучше, если вы выберете новое имя для метода в производном классе.

3.8. Имена пространств имен и типов

В некоторых контекстах в программе C# требуется, чтобы *имя-пространства-имен* или *имя-типа* были определены.

```
имя-пространства-имен:
   имя-пространства-имен-или-типа

имя-типа:
   имя-пространства-имен-или-типа

имя-пространства-имен-или-типа:
   идентификатор список-аргументов-типа_opt
   имя-пространства-имен-или-типа   .  идентификатор список-аргументов-типа_opt
   уточненный-псевдоним-элемента
```

Имя-пространства-имен есть *имя-пространства-имен-или-типа*, которое относится к пространству имен. Согласно правилам, описанным ниже, *имя-пространства-имен-или-типа имени-пространства-имен* должно относиться к пространству имен, в противном случае возникает ошибка компиляции. Никакие аргументы-типы (раздел 4.4.1) не могут присутствовать в *имени-пространства-имен* (аргументы-типы могут быть только у типов).

Имя-типа есть *имя-пространства-имен-или-типа*, которое относится к типу. Согласно правилам, описанным ниже, *имя-пространства-имен-или-типа имени-типа* должно относиться к *имени-типа*, в противном случае возникает ошибка компиляции.

Если *имя-пространства-имен-или-типа* представляет собой *уточненный-псевдоним-элемента*, его значение соответствует описанному в разделе 9.7.

Иначе *имя-пространства-имен-или-типа* может быть представлено в одной из четырех форм:

- `I`
- $I<A_1, ..., A_K>$
- `N.I`
- $N.I<A_1, ..., A_K>$

где `I` — это одиночный идентификатор, `N` — *имя-пространства-имен-или-типа*, $<A_1, ..., A_K>$ — необязательный *список-аргументов-типа*. Когда *список-аргументов-типа* не задан, считается, что `K` равно нулю.

Значение *имени-пространства-имен-или-типа* определяется следующим образом:

- Если *имя-пространства-имен-или-типа* существует в форме `I` или $I<A_1, ..., A_K>$:
  - Если `K` равно нулю и *имя-пространства-имен-или-типа* появляется в объявлении обобщенного метода (раздел 10.6) и если это объявление включает в себя параметр-тип (раздел 10.1.3) с именем `I`, то это *имя-пространства-имен-или-типа* относится к этому параметру-типу.
  - Иначе, если *имя-пространства-имен-или-типа* появляется внутри объявления типа, то для каждого конкретного типа `T` (раздел 10.3.1), начиная с конкрет-

ного типа в этом объявлении типа и продолжая конкретными типами в каждом объявлении объемлющего класса или структуры (если таковые имеются):

- Если `K` равно нулю и объявление `T` содержит параметр-тип с именем `I`, то *имя-пространства-имен-или-типа* относится к этому параметру-типу.
- Иначе, если *имя-пространства-имен-или-типа* появляется внутри тела объявления типа и `T` или любой из его базовых типов содержит вложенный доступный тип с именем `I` и `K` параметрами-типами, тогда *имя-пространства-имен-или-типа* относится к типу, сконструированному с данными аргументами-типами. Если таких типов более одного, выбирается тип, объявленный внутри наиболее производного типа. Заметим, что элементы, которые не являются типами (константы, поля, методы, свойства, индексаторы, операции, конструкторы экземпляра, деструкторы и статические конструкторы) и элементы-типы с отличающимся числом параметров-типов при определении значения *имени-пространства-имен-или-типа* во внимание не принимаются.
- Если предыдущие шаги были безуспешны, то для каждого пространства имен `N`, начиная с пространства имен, в котором встречается *имя-пространства-имен-или-типа*, продолжая каждым объемлющим пространством имен (если таковые существуют) и заканчивая глобальным пространством имен, выполняются следующие шаги до тех пор, пока сущность не будет определена:
- Если `K` равно нулю и `I` есть имя пространства имен в `N`, тогда:
  - Если *имя-пространства-имен-или-типа* находится в окружении объявления пространства имен для `N` и это объявление пространства имен содержит *директиву-внешнего-псевдонима* или *using-директиву-псевдонима*, которая связывает имя `I` с пространством имен или типом, тогда *имя-пространства-имен-или-типа* неоднозначно и выдается ошибка компиляции.
  - В противном случае *имя-пространства-имен-или-типа* относится к пространству имен `I` в `N`.
- Иначе, если `N` содержит доступный тип с именем `I` и `K` параметрами-типами, то:
  - Если `K` равно нулю и *имя-пространства-имен-или-типа* находится в окружении объявления пространства имен для `N` и это объявление пространства имен содержит *директиву-внешнего-псевдонима* или *using-директиву-псевдонима*, которая связывает имя `I` с пространством имен или типом, то *имя-пространства-имен-или-типа* неоднозначно и выдается ошибка компиляции.
  - В противном случае *имя-пространства-имен-или-типа* относится к типу, сконструированному с данными аргументами-типами.
- Иначе, если *имя-пространства-имен-или-типа* находится в окружении объявления пространства имен для `N`:
  - Если `K` равно нулю и объявление пространства имен содержит *директиву-внешнего-псевдонима* или *using-директиву-псевдонима*, кото-

рая связывает имя I с импортированным пространством имен или типом, то *имя-пространства-имен-или-типа* относится к этому пространству имен или типу.

- Иначе, если пространство имен, импортированное с помощью *using-директивы-пространства-имен* в объявлении пространства имен, содержит в точности один тип с именем `I` и `K` параметрами-типами, то *имя-пространства-имен-или-типа* относится к типу, сконструированному с данными аргументами-типами.
- Иначе, если пространство имен, импортированное при помощи *using-директивы-пространства-имен* в объявлении пространства имен, содержит более одного типа с именем `I` и `K` параметрами-типами, то *имя-пространства-имен-или-типа* неоднозначно и выдается ошибка.
- Иначе *имя-пространства-имен-или-типа* не определено и выдается ошибка компиляции.

• Иначе, *имя-пространства-имен-или-типа* существует в форме `N.I` или $N.I<A_1, ..., A_K>$. Сначала разрешается `N` как *имя-пространства-имен-или-типа*. Если разрешение `N` не удалось, выдается ошибка компиляции. В противном случае `N.`I или $N.I<A_1, ..., A_K>$ разрешается следующим образом:

Если `K` равно нулю и `N` относится к пространству имен и `N` содержит вложенное пространство имен с именем `I`, то *имя-пространства-имен-или-типа* относится к этому вложенному пространству имен.

Иначе, если `N` относится к пространству имен и `N` содержит вложенное пространство имен с именем `I` и `K` параметрами-типами, то *имя-пространства-имен-или-типа* относится к типу, сконструированному с данными аргументами-типами.

Иначе, если `N` относится к типу класса или структуры (возможно, сконструированному) и при этом `N` или любой из его родительских классов содержат вложенный доступный тип с именем `I` и `K` параметрами-типами, то *имя-пространства-имен-или-типа* относится к этому типу, сконструированному с данными аргументами-типами. Если имеется более одного такого типа, выбирается наиболее производный тип. Заметим, что если значение `N.I` определяется как часть разрешения спецификации родительского класса `N`, то непосредственным базовым классом `N` считается `object` (раздел 10.1.4.1).

Иначе `N.I` является недопустимым *именем-пространства-имен-или-типа* и выдается ошибка компиляции.

ВЛАДИМИР РЕШЕТНИКОВ

Этот алгоритм отличается от соответствующего алгоритма для простых имен (раздел 7.6.2), хотя в то же время и похож на него. Это означает, что в хитрых случаях один и тот же идентификатор может иметь различные значения в контекстах *имени-типа* и *простого-имени*:

```
class T
{
   public const int X = 1;
}
```

```
class C
{
   void Foo<T>(int x = T.X /* global::T */,
      T y = default(T) /* type parameter */) { }
}
```

Имя-пространства-имен-или-типа может относиться к статическому классу (раздел 10.11.3) только в следующих случаях:

- *Имя-пространства-имен-или-типа* есть `T` в *имени-пространства-имен-или-типа* в форме `T.I`, или
- *Имя-пространства-имен-или-типа* есть `T` в *выражении-typeof* (раздел 7.5.11) в форме `typeof(T)`.

3.8.1. Полные имена

Каждое пространство имен и каждый тип имеют **полное имя**, которое однозначно определяет это пространство имен или тип среди всех других. Полное имя пространства имен или типа `N` определяется следующим образом:

- Если `N` является элементом глобального пространства имен, его полное имя есть `N`.
- В противном случае его полное имя есть `S.N`, где `S` является полным именем пространства имен или типа, в котором объявлено имя `N`.

Иными словами, полное имя `N` есть полный иерархический путь идентификаторов, который ведет к `N`, начиная с глобального пространства имен. Поскольку каждый элемент пространства имен или типа должен иметь уникальное имя, отсюда следует, что полное имя пространства имен или типа всегда уникально.

Пример, приведенный ниже, показывает некоторые объявления пространств имен и типов с их полными именами.

```
class A { }                 // A

   namespace X              // X
    {
       class B              // X.B
       {
           class C { }      // X.B.C
       }
       namespace Y          // X.Y
       {
           class D { }      // X.Y.D
       }
    }

    namespace X.Y           // X.Y
    {
       class E { }          // X.Y.E
    }
```

ДЖОЗЕФ АЛЬБАХАРИ

Если полное имя конфликтует с частично уточненным именем или с неуточненным именем (вложенного доступного типа, например), последнее побеждает. Чтобы полное имя победило, нужно снабдить его префиксом `global::` (раздел 9.7). В коде, написанном людьми, такие коллизии возникают крайне редко; в машинном коде это далеко не так. По этой причине некоторые генераторы кода в средствах разработки и интегрированных средах разработки (IDE) указывают префикс `global::` перед всеми полными именами типа, чтобы устранить любую возможность конфликта.

3.9. Автоматическое управление памятью

C# обеспечивает автоматическое управление памятью, которое освобождает разработчиков от размещения объектов и освобождения памяти вручную. Политика автоматического управления памятью реализована с помощью **сборщика мусора**. Жизненный цикл управления памятью объекта выглядит следующим образом:

1. Когда объект создается, для него выделяется память, запускается конструктор, и объект рассматривается как существующий.

2. Если к объекту или его части невозможно получить доступ при любом возможном продолжении выполнения программы, иному чем запуск деструкторов, объект считается более не используемым и становится кандидатом на уничтожение. Компилятор C# и сборщик мусора могут решить проанализировать код для того, чтобы определить, какие ссылки на объект могут использоваться в будущем. Например, если локальная переменная, которая находится в области видимости, является единственной ссылкой на объект, но на эту локальную переменную никогда не ссылаются при любом возможном продолжении выполнения из текущей точки выполнения процедуры, сборщик мусора может (но не обязан) посчитать объект более не используемым.

КРИСТИАН НЕЙГЕЛ

Из деструктора компилятор C# создает код для переопределения метода `Finalize` родительского класса. Переопределение метода `Finalize` также означает избыточные расходы на создание объекта и сохраняет объект до запуска финализатора. В C++/CLI код, сгенерированный деструктором, выполняется интерфейсом `IDisposable`, который более приспособлен для детерминированной очистки памяти.

ДЖОН СКИТ

Момент, в который объект станет подлежащим деструкции, может наступить раньше, чем вы ожидаете. В частности, деструктор объекта может работать, пока другой поток все еще выполняет метод экземпляра «в» том же самом объекте — если метод экзем-

> пляра не ссылается на любую переменную экземпляра в любой возможной ветви кода после текущей точки выполнения.
> К счастью, такое поведение может порождать сколько-нибудь заметные проблемы только в типах с деструкторами, которые используются в коде .NET относительно редко с момента появления `SafeHandle`.

3. После того как объект стал подлежащим деструкции, через неопределенное время (раздел 10.13) будет запущен деструктор объекта (если таковой существует). Деструктор объекта запускается только один раз, если это не переопределено путем его явных вызовов.

> **КРИСТИАН НЕЙГЕЛ**
>
> «Деструкция происходит спустя неопределенное время...» Когда создается деструктор, неплохо также реализовать интерфейс `IDisposable`. При реализации этого интерфейса код очистки никогда не будет вызван раньше, чем объект станет не нужен.

> **КРИСТИАН НЕЙГЕЛ**
>
> Чтобы вызвать деструктор более одного раза, `GC.ReRegisterForFinalize` сохраняет объект живым и позволяет вызвать деструктор еще раз. Вместо того чтобы латать дыры таким образом, лучше изменить архитектуру приложения.

4. Когда деструктор объекта запущен, если объект или любая его часть не могут быть доступны для любого возможного продолжения выполнения, в том числе и для выполнения деструктора, объект рассматривается как недоступный и подлежит сборке мусора.

5. Наконец, через некоторое время после того, как объект становится подлежащим сборке мусора, сборщик мусора освобождает память, связанную с этим объектом.

Сборщик мусора сохраняет информацию об использовании объекта и использует эту информацию для принятия решений по управлению памятью: где в памяти расположить вновь созданный объект, когда переместить объект и когда объект является более не используемым или недоступным.

Подобно другим языкам, которые предполагают существование сборщика мусора, C# сконструирован так, что сборщик мусора может реализовывать широкий спектр политик управления памятью. Например, в C# нет требований, чтобы деструктор был запущен или чтобы объекты были удалены, как только это станет возможным, или чтобы деструкторы запускались в некотором определенном порядке или в каком-то определенном потоке.

Поведение сборщика мусора может до некоторой степени управляться с помощью статических методов класса `System.GC`. Этот класс можно использовать, чтобы запустить процесс сборки мусора, запустить (или не запускать) деструкторы и т. д.

ЭРИК ЛИППЕРТ

Использование этих статических методов для управления поведением сборщика мусора практически никогда не является хорошим решением. При генерации кода очень высока вероятность того, что сборщик мусора лучше, чем ваша программа, знает, когда заняться удалением объектов.

Явная настройка сборщика мусора во время выполнения программы допустима разве что в целях тестирования приложения.

Так как сборщику мусора предоставлены широкие полномочия в решении вопроса о сборке объектов и запуске деструкторов, соответствующая реализация может выдать в результате код, отличающийся от приведенного ниже. Программа

```
using System;
    class A
    {
        ~A()
        {
            Console.WriteLine("Уничтожается экземпляр A");
        }
    }
    class B
    {
        object Ref;
        public B(object o)
        {
            Ref = o;
        }
        ~B()
        {
            Console.WriteLine("Уничтожается экземпляр B");
        }
    }
    class Test
    {
        static void Main()
        {
            B b = new B(new A());
            b = null;
            GC.Collect();
            GC.WaitForPendingFinalizers();
        }
    }
```

создает экземпляр класса `A` и экземпляр класса `B`. Эти объекты становятся подлежащими сборке мусора, когда переменной `b` присваивается значение `null`, так как с этого времени любой написанный пользователем код не может получить к ним доступ. Вывод будет либо

```
Уничтожается экземпляр A
Уничтожается экземпляр B
```

либо

```
Уничтожается экземпляр B
Уничтожается экземпляр A
```

поскольку в языке не существует никаких ограничений на порядок, в котором сборщик мусора собирает объекты.

В некоторых особых случаях различие между «подлежащий деструкции» и «подлежащий сборке мусора» может быть существенным. Например:

```
using System;
using System;
class A
{
    ~A() {
        Console.WriteLine("Уничтожается экземпляр A");
    }
    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}
class B
{
    public A Ref;
    ~B() {
        Console.WriteLine("Уничтожается экземпляр B");
        Ref.F();
    }
}
class Test
{
    public static A RefA;
    public static B RefB;
static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;
        // A и B теперь подлежат деструкции
        GC.Collect();
        GC.WaitForPendingFinalizers();
        // B теперь подлежит сборке мусора, но A - нет
        if (RefA != null)
            Console.WriteLine("RefA не null");
}
```

Если сборщик мусора выберет в этой программе вызов деструктора `A` до вызова деструктора `B`, программа выведет следующее:

```
Уничтожается экземпляр A
Уничтожается экземпляр B
A.F
RefA не null
```

Отметим, что хотя экземпляр `A` не используется и деструктор `A` запущен, все еще можно вызвать методы для `A` (в данном случае метод `F`) из другого деструктора. Отметим также, что выполнение деструктора может снова сделать объект используемым из главной программы. В данном случае выполнение деструктора `B` приводит экземпляр `A`, который прежде не использовался, к тому, что он становится доступ-

ным из «живой» ссылки `Test.RefA`. После вызова `WaitForPendingFinalizers` экземпляр `B` подлежит сборке, но экземпляр `A` не подлежит из-за ссылки `Test.RefA`.

Для предотвращения неразберихи и непредсказуемого поведения рекомендуется, чтобы деструкторы выполняли только очистку данных, хранящихся в собственных полях их объектов, и не выполняли бы действий над объектами, на которые они ссылаются, или над статическими полями.

ЭРИК ЛИППЕРТ

Есть даже лучшая идея: чтобы деструкторы выполняли только очистку полей, содержащих данные, представляющие неуправляемые объекты, такие как дескрипторы операционной системы. Поскольку мы не знаем, в каком потоке запустится деструктор и когда он запустится, особенно важно, чтобы деструктор производил как можно меньше побочных эффектов.

Вызов `Console.WriteLine` в приведенном выше примере грубо пренебрегает хорошим советом не производить иных действий, кроме очистки. Такой код приведен исключительно в целях обучения. В реальном коде деструкторы никогда не должны пытаться сделать что-нибудь, имеющее сложное побочное действие, такое, как, например, вывод на консоль.

Альтернативой использованию деструкторов может служить реализация в классе интерфейса `System.IDisposable`. Она позволяет клиенту объекта определять, когда освобождать ресурсы объекта, обычно через получение доступа к объекту как к ресурсу в операторе `using` (раздел 8.13).

БРЭД АБРАМС

В девяти случаях из десяти использование `GC.Collect()` является ошибкой. Чаще всего оно указывает на убогую конструкцию, кое-как собранную воедино. Сборщик мусора — это инструмент с тонкой настройкой, подобный автомобилю «порше». Не стоит парковаться «по звуку», сидя за рулем новенького «порше», и аналогично не стоит вмешиваться в алгоритм сборки мусора. Сборщик мусора сконструирован таким образом, чтобы ненавязчиво в нужное время освобождать наиболее важную неиспользуемую память. Вмешательство в его деятельность при вызове `GC.Collect()` может нарушить баланс и настройку. Прежде чем прибегнуть к такому средству, потратьте несколько минут на то, чтобы уяснить, что в действительности вам требуется. Где расположены все ваши экземпляры? Там ли потерянные ссылки, где они, по-вашему, должны находиться? В нужных ли местах вы используете слабые ссылки?

КШИШТОФ ЦВАЛИНА

Некоторые люди связывают проблемы производительности в современных системах, основанных на виртуальных машинах, с использованием сборки мусора. На самом деле современные сборщики мусора так эффективны, что осталось очень мало программного обеспечения, в котором возникают проблемы собственно при выполнении сборки мусора. Самые главные виновники данной ситуации — плохое проектирование приложений и, к сожалению, многие библиотеки каркаса.

КРИС СЕЛЛЗ

В современных бизнес-системах, для построения которых в основном используется .NET, сборка мусора обеспечивает удивительную устойчивость. Только сейчас, когда мы начинаем использовать управляемый код в приложениях реального времени, таких как игры (например, XNA для Xbox) и платформы Windows Phone 7, внимательное исследование поведения сборщика мусора становится важным. Даже в таких случаях эффективное проектирование при распределении ресурсов значит намного больше, чем трюки непосредственно со сборщиком мусора.

КРИСТИАН НЕЙГЕЛ

В дополнение к высказыванию Кшиштофа: обсуждения по поводу производительности сборщика мусора обычно напоминают давнишние сравнения кода C с кодом ассемблера или несколько более поздние, когда производительность C++ сравнивали с C. Конечно, все еще есть место и для кода ассемблера, и для кода C, но большинство приложений прекрасно себя чувствует в управляемой среде выполнения.

БИЛЛ ВАГНЕР

В общем, эти замечания свидетельствуют о том, как редко ваши обычные предположения относительно деструктора оказываются верными. Может оказаться, что деструкторы элементов уже выполнились. Они могут быть вызваны в различных потоках, поэтому локальная память потока может оказаться недействительной. Они вызываются системой, поэтому ваше приложение не увидит сообщений об ошибках деструкторов, использующих исключения. Трудно преувеличить степень защищенности, с которой вы должны писать деструкторы. К счастью, они нужны только в редких случаях.

3.10. Порядок выполнения

Выполнение программы C# происходит таким образом, что побочные эффекты каждого потока выполнения сохраняются в критических точках выполнения. *Побочный эффект* определяется как чтение или запись асинхронно-изменяемого поля, запись в переменную, не изменяемую асинхронно, запись во внешний ресурс и выбрасывание исключения. Критические точки выполнения, в которых порядок этих побочных эффектов должен быть сохранен, — это ссылки на асинхронно-изменяемые поля (раздел 10.5.3), операторы `lock` (раздел 8.12), а также создание и уничтожение потоков. Окружение может менять порядок выполнения программы C#, при этом подчиняясь следующим ограничениям:

- Зависимость по данным сохраняется внутри потока выполнения. То есть значение каждой переменной вычисляется так, как если бы все операторы в потоке были выполнены в первоначально установленном в программе порядке.
- Правила, касающиеся порядка инициализации, сохраняются (разделы 10.5.4 и 10.5.5).

- Порядок побочных эффектов сохраняется в отношении временного чтения (volatile reads) и записи (раздел 10.5.3). Добавим, что нет необходимости для среды выполнения вычислять часть выражения, если можно установить, что значение этого выражения не используется и что производятся ненужные побочные эффекты (в том числе порождаемые вызовом метода или доступом к асинхронно-изменяемому полю). Когда выполнение программы прерывается асинхронным событием (таким как исключение, выброшенное другим потоком), не гарантировано, что наблюдаемые побочные эффекты будут появляться в первоначально установленном программой порядке.

Глава 4
Типы

Типы в языке C# делятся на две главные категории: **типы-значения** и **ссылочные типы**. Типы обеих категорий могут быть **обобщенными**, то есть имеющими один или более **параметров-типов**. Параметры-типы могут быть и типа-значения, и ссылочного типа.

тип:
  тип-значение
  ссылочный-тип
  параметр-тип

Третья категория типов — указатели. Они доступны только в небезопасном коде. Этот вопрос будет обсуждаться в разделе 18.2.

Типы-значения отличаются от ссылочных типов тем, что переменные типов-значений содержат непосредственно данные, тогда как переменные ссылочных типов хранят *ссылки* на данные (*объекты*). Для ссылочных типов допустимо, чтобы две переменные ссылались на один и тот же объект, и, таким образом, возможно, что операции над одной переменной действуют на объект, на который ссылается другая переменная. Для типов-значений каждая переменная имеет свою копию данных, так что операция над одной переменной никак не может воздействовать на другую переменную.

Система типов C# унифицирована таким образом, что значение любого типа может рассматриваться как объект. Каждый тип C# прямо или опосредованно наследуется от класса `object`, который является корневым родительским классом для всех типов. Значения ссылочных типов могут непосредственно рассматриваться как объекты. Значения типов-значений могут использоваться как объекты при помощи операций упаковки (boxing) и распаковки (unboxing) (раздел 4.3).

ЭРИК ЛИППЕРТ

Обычно мы не думаем про интерфейсы или типы параметров-типов, что они имеют «родительский класс». Хочется подчеркнуть, что каждый конкретный объект независимо от того, как он представлялся при компиляции, во время выполнения программы может использоваться как экземпляр класса `object`.

4.1. Типы-значения

К типам-значениям относятся структурные типы и перечисления. В C# имеется набор предопределенных структурных типов, называемых *простыми типами*. Простые типы идентифицируются с помощью зарезервированных слов.

```
тип-значение:
   структура
   перечисление

структура:
   имя-типа
   простой-тип
   обнуляемый-тип

простой-тип:
   арифметический-тип
   bool

арифметический-тип:
   целочисленный-тип
   вещественный-тип
   decimal

целочисленный-тип:
   sbyte
   byte
   short
   ushort
   int
   uint
   long
   ulong
   char

вещественный-тип:
   float
   double

обнуляемый-тип:
   необнуляемый-тип-значение   ?

необнуляемый-тип-значение:
   тип

перечисление:
   имя-типа
```

В отличие от переменных ссылочного типа, переменная типа-значения может содержать значение `null`, только если ее тип является обнуляемым (nullable). Для каждого необнуляемого типа-значения существует соответствующий обнуляемый тип-значение, включающий тот же самый ряд значений плюс значение `null`.

Присваивание переменной типа-значения создает *копию* присвоенного значения. Это отличает его от присваивания переменной ссылочного типа, когда копируется ссылка, а не определяемый ею объект.

4.1.1. Тип System.ValueType

Все типы-значения неявно наследуются от класса `System.ValueType`, который, в свою очередь, является наследником класса `object`. Никакой тип не может быть

унаследован от типа-значения, и, таким образом, все типы-значения неявным образом являются бесплодными (раздел 10.1.1.2).

Заметим, что сам `System.ValueType` не является *типом-значением*. Напротив, он является *классом*, от которого автоматически наследуются все типы-значения.

> **ЭРИК ЛИППЕРТ**
>
> Эта особенность часто сбивает с толку новичков. Меня часто спрашивают: «Но как это возможно, чтобы тип-значение наследовался от ссылочного типа?» Я думаю, что такое замешательство является результатом непонимания того, что означает «унаследованные». Наследование не означает, что где-то в памяти, занимаемой потомком, можно найти такое же расположение двоичных разрядов, как в его родительском типе. Оно просто говорит о том, что существует некоторый механизм, с помощью которого к элементам родительского типа можно получить доступ из типа-потомка.

4.1.2. Конструкторы по умолчанию

Во всех типах-значениях неявно объявлен открытый конструктор экземпляра без параметров, называемый *конструкторами по умолчанию*. Конструктор по умолчанию возвращает инициализированный нулями экземпляр, называемый *значением по умолчанию* этого типа:

- Для всех *простых-типов* значением по умолчанию является значение битовой комбинации, состоящей из всех нулей:
  - Для типов `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong` значением по умолчанию является `0`.
  - Для типа `char` значением по умолчанию является `'\x0000'`.
  - Для типа `float` значением по умолчанию является `0.0f`.
  - Для типа `double` значением по умолчанию является `0.0d`.
  - Для типа `decimal` значением по умолчанию является `0.0m`.
  - Для типа `bool` значением по умолчанию является `false`.
  - Для *перечисления* `E` значением по умолчанию является `0`, преобразованный к типу `E`.
  - Для *структурного-типа* значением по умолчанию является значение, полученное при присваивании всем полям типа-значения их значений по умолчанию, а всем полям ссылочного типа — значения `null`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Очевидно, что выражение «все поля» означает здесь только поля экземпляров (то есть не статические поля). Сюда также относятся подобные полям события экземпляров, если таковые имеются.

- Для *обнуляемого-типа* значение по умолчанию есть экземпляр, для которого свойство `HasValue` имеет значение `false` и свойство `Value` не определено. Значение по умолчанию обнуляемого типа называется *нулевым значением* обнуляемого типа.

Подобно другим конструкторам экземпляра, конструктор по умолчанию типа-значения вызывается при помощи операции `new`. По соображениям эффективности это требование не означает, что реализация должна действительно генерировать вызов конструктора. В следующем примере переменные `i` и `j` инициализированы нулевыми значениями.

```
class A
{
   void F() {
      int i = 0;
      int j = new int();
   }
}
```

Поскольку каждый тип-значение неявно содержит открытый конструктор экземпляра без параметров, структурный тип не может содержать явного объявления конструктора без параметров, однако конструкторы экземпляров с параметрами в структурном типе объявлять допускается (раздел 11.3.8).

ЭРИК ЛИППЕРТ

Другой хороший способ получить значение типа по умолчанию — использовать выражение `default(type)`.

ДЖОН СКИТ

Это один из примеров, который показывает, что язык C# и базовая платформа могут использовать различные принципы. Если вы запросите у платформы .NET конструкторы типа-значения, обычно вы не найдете конструкторов без параметров. Вместо этого в .NET есть особая инструкция для инициализации типа-значения значением по умолчанию. Обычно это небольшое несоответствие не имеет особого значения для разработчиков, но полезно знать, что такое может быть и что это не является недостатком какой-либо из данных спецификаций.

4.1.3. Структурные типы

Структурным типом является тип-значение, в котором можно объявлять константы, поля, методы, свойства, индексаторы, операции, конструкторы экземпляров, статические конструкторы и вложенные типы. Объявление структурных типов описано в разделе 11.1.

4.1.4. Простые типы

C# поддерживает ряд предопределенных структурных типов, называемых *простыми типами*. Простые типы идентифицируются с помощью зарезервированных слов, которые являются просто псевдонимами для предопределенных структурных типов в пространстве имен `System`, как показано в следующей таблице.

Зарезервированное слово	**Соответствующий тип**
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`
ushort	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

Поскольку простой тип является псевдонимом структурного типа, каждый простой тип содержит элементы. Например, тип `int` содержит элементы, объявленные в `System.Int32`, и элементы, унаследованные от `System.Object`, и следующие операторы являются разрешенными:

```
int i = int.MaxValue;          // константа System.Int32.MaxValue
string s = i.ToString();       // метод экземпляра
System.Int32.ToString()
string t = 123.ToString();     // метод экземпляра
System.Int32.ToString()
```

Простые типы отличаются от других структурных типов тем, что в них разрешены некоторые дополнительные операции:

- Большинство простых типов позволяет создавать значение путем записи *литералов* (раздел 2.4.4). Например, 123 является литералом типа `int`, `'a'` является константой типа `char`. C#, вообще говоря, не делает заготовок для литералов структурных типов, и значения не по умолчанию других структурных типов в конечном счете всегда создаются с помощью конструкторов экземпляров этих структурных типов.

ЭРИК ЛИППЕРТ

Слово «большинство» в фразе «большинство простых типов» означает, что некоторые простые типы, например `short`, не имеют константной формы. Любой целочисленный литерал, достаточно малый для представления в виде `short`, неявно преобразуется в `short`, когда используется как этот тип, так что в этом смысле константные значения для всех простых типов существуют.

продолжение ✐

> Для простых типов существует несколько возможных значений, не имеющих литеральной формы. Например, это значения NaN (Not-a-Number) для типов с плавающей точкой.

- Когда все операнды выражения являются константами простых типов, компилятор может вычислить значение выражения во время компиляции. Такое выражение называется *константным-выражением* (раздел 7.19). Выражения, содержащие операции, определяемые другими структурными типами, не могут рассматриваться как константные выражения.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Уточнение: компилятор не «может вычислить», а **всегда** полностью вычисляет *константные-выражения* во время компиляции.

С помощью `const` можно объявлять константы только простых типов (раздел 10.4). Невозможно создать константы других структурных типов, но похожий эффект могут обеспечить поля `static readonly`.

- Преобразования, в которых участвуют простые типы, могут использоваться при выполнении преобразований, определяемых другими структурными типами, но операции преобразования, определенные пользователем, никогда не могут использоваться при выполнении других определенных пользователем операций (раздел 6.4.3).

> **ДЖОЗЕФ АЛЬБАХАРИ**
> Простые типы также обеспечивают средства, с помощью которых компилятор может использовать прямую поддержку целочисленных вычислений и вычислений с плавающей точкой в IL (и, в конечном счете, в процессоре). Такая схема позволяет выполнять арифметические операции над простыми типами, которые поддерживаются процессором (обычно `float`, `double` и целые) со скоростью, присущей данной среде.

4.1.5. Целочисленные типы

C# поддерживает девять целочисленных типов: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` и `char`. Целочисленные типы имеют следующие размеры и интервалы значений:

- Тип `sbyte` представляет 8-разрядные целые значения со знаком в интервале от –128 до 127.
- Тип `byte` представляет 8-разрядные целые значения без знака в интервале от 0 до 255.
- Тип `short` представляет 16-разрядные целые значения со знаком в интервале от –32 768 до 32 767.

- Тип `ushort` представляет 16-разрядные целые значения без знака в интервале от 0 до 65 535.
- Тип `int` представляет 32-разрядные целые значения со знаком в интервале от –2 147 483 648 до 2 147 483 647.
- Тип `uint` представляет 32-разрядные целые значения без знака в интервале от 0 до 4 294 967 295.
- Тип `long` представляет 64-разрядные целые значения со знаком в интервале от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807.
- Тип `ulong` представляет 64-разрядные целые значения без знака в интервале от 0 до 18 446 744 073 709 551 615.
- Тип `char` представляет 16-разрядные целые значения без знака в интервале от 0 до 65 535. Ряд возможных значений для типа `char` соответствует последовательности символов Unicode. Хотя `char` имеет такое же представление, как `ushort`, не все операции, разрешенные для одного типа, разрешены для другого.

> **ДЖЕСС ЛИБЕРТИ**
>
> Должен признаться, что, учитывая мощь современных персональных компьютеров и то, что время программиста стоит дороже, чем память компьютера, я склоняюсь к тому, чтобы использовать `int` почти для всех целочисленных (не вещественных) значений и `double` для любых вещественных значений. Все остальное я чаще всего игнорирую.

Унарные и бинарные операции для целого типа всегда выполняются с 32-разрядной точностью со знаком или без знака или с 64-разрядной точностью со знаком или без знака:

- Для унарных операций `+` и `~` операнды приводятся к типу `T`, где `T` есть первый из типов `int`, `uint`, `long` и `ulong`, который может полностью представить все возможные значения операндов. Операция, которая затем выполняется, будет выполняться с точностью, определяемой типом `T`, и тип результата будет `T`.
- Для унарной операции `–` операнд преобразуется к типу `T`, где `T` есть первый из типов `int` и `long`, который может полностью представить возможные значения операнда. Операция, которая затем выполняется, будет выполняться с точностью, определяемой типом `T`, и тип результата будет `T`. Унарная операция `–` не может применяться к операндам типа `ulong`.
- Для бинарных операций `+`, `–`, `*`, `/`, `%`, `&`, `^`, `|`, `==`, `!=`, `>`, `<`, `>=` и `<=` операнды преобразуются к типу `T`, где `T` есть первый из типов `int`, `uint`, `long` и `ulong`, который может полностью представить возможные значения операндов. Операция, которая затем выполняется, будет выполняться с точностью, определяемой типом `T`, и тип результата будет `T` (или `bool` для операций отношения). Для бинарных операций недопустимо, чтобы один операнд имел тип `long`, а другой имел тип `ulong`.
- Для бинарных операций `<<` и `>>` левый операнд приводится к типу `T`, где `T` есть первый из типов `int`, `uint`, `long` и `ulong`, который может полностью

представить все возможные значения операнда. Операция, которая затем выполняется, будет выполняться с точностью, определяемой типом `T`, и тип результата будет `T`.

Тип `char` классифицируется как целочисленный тип, но он отличается от других целочисленных типов двумя особенностями:

- Не существует неявного преобразования других типов к типу `char`. В частности, хотя типы `sbyte`, `byte` и `ushort` и имеют интервал значений, который может быть полностью представлен с использованием типа `char`, не существует неявного преобразования типов `sbyte`, `byte` и `ushort` к типу `char`.
- Константы типа `char` должны быть записаны как *символьные-литералы* или как *целые-литералы* в сочетании с приведением к типу `char`. Например, `(char)10` — то же самое, что и `'\x000A'`.

Операции и операторы `checked` и `unchecked` используются для управления контекстом контроля переполнения при выполнении арифметических операций и преобразований целого типа (раздел 7.6.12). В проверяемом контексте переполнение приводит к ошибке компиляции или к выбрасыванию исключения `System.OverflowException`. В непроверяемом контексте переполнение игнорируется, а старшие биты, которые не могут быть представлены в типе назначения, отбрасываются.

4.1.6. Типы с плавающей точкой

C# поддерживает два типа с плавающей точкой: `float` и `double`. В типах `float` и `double` используется соответственно 32-разрядная точность и 64-разрядная двойная точность в формате IEEE 754, который обеспечивает следующее множество значений:

- Положительный ноль и отрицательный ноль. В большинстве ситуаций положительный ноль и отрицательный ноль ведут себя так же, как обычный ноль, но некоторые операции их различают (раздел 7.8.2).

ВЛАДИМИР РЕШЕТНИКОВ

Полезно знать, что реализация метода `Equals` по умолчанию в типах-значениях может в некоторых случаях использовать поразрядное сравнение для ускорения выполнения. Если два экземпляра типа-значения содержат в своих полях соответственно положительный и отрицательный ноль, при сравнении они могут оказаться не равными. Для изменения поведения по умолчанию можно использовать переопределение метода `Equals`.

```
using System;
struct S
{
   double X;
   static void Main()
```

```
   {
      var a = new S {X = 0.0};
      var b = new S {X = -0.0};
      Console.WriteLine(a.X.Equals(b.X)); // True
      Console.WriteLine(a.Equals(b));     // False
   }
}
```

ПИТЕР СЕСТОФТ

Некоторое недоразумение, связанное с отрицательным нулем, может возникать из того факта, что существующие в настоящее время реализации C# печатают положительный и отрицательный ноль одинаково, как 0.0, и на этот способ отображения не влияет никакая комбинация параметров форматирования. Хотя, возможно, это сделано из лучших побуждений, это неудачное решение. Чтобы обнаружить отрицательный ноль, нужно использовать странноватый код, подобный приведенному ниже, который работает, поскольку `1/(-0.0) = -Infinity < 0`:

```
public static string DoubleToString(double d) {
   if (d == 0.0 && 1/d < 0)
      return "-0.0";
   else
      return d.ToString();
}
```

- Положительная бесконечность и отрицательная бесконечность. Бесконечности порождаются такими операциями, как деление отличного от нуля числа на ноль. Например, `1.0/0.0` дает положительную бесконечность, а `–1.0/0.0` дает отрицательную бесконечность.
- Значение, не являющееся числом (*Not-a-Number*), для него часто используется сокращение NaN. Это значение является результатом недопустимых операций с плавающей точкой, таких как деление нуля на нуль.

ПИТЕР СЕСТОФТ

Существует большое количество отдельных NaN, каждый из которых несет свою «полезную нагрузку». См. комментарии к разделу 7.8.1.

- Конечная последовательность ненулевых значений в форме $s \times m \times 2^e$, где s равно 1 или –1, а m и e определяются конкретным типом с плавающей точкой: для `float` $0 < m < 2^{24}$ и $-149 \le e \le 104$; для `double` $0 < m < 2^{53}$ и $-1075 \le e \le 970$. Ненормализованные значения с плавающей точкой рассматриваются как допустимые ненулевые значения.

Тип `float` может представлять значения приблизительно от 1.5×10^{-45} до 3.4×10^{38} с точностью 7 цифр.

Тип `double` может представлять значения приблизительно от 5.0×10^{-324} до 1.7×10^{308} с точностью 15 или 16 цифр.

Если один из операндов бинарной операции относится к типу с плавающей точкой, другой операнд должен быть целочисленным или с плавающей точкой, и операция вычисляется следующим образом:

- Если один из операндов целого типа, этот операнд приводится к типу с плавающей точкой другого операнда.
- Затем, если любой из операндов имеет тип `double`, другой операнд приводится к типу `double`, операция выполняется с интервалом значений и точностью по меньшей мере `double`, и тип результата есть `double` (или `bool` для операций отношения).
- В остальных случаях операция выполняется с интервалом значений и точностью по меньшей мере `float`, и тип результата есть `float` (или `bool` для операций отношения).

Операции с плавающей точкой, в том числе операции присваивания, никогда не приводят к генерации исключений. Вместо этого в исключительных ситуациях операции с плавающей точкой дают в результате ноль, бесконечность или NaN, как описано ниже:

- Если результат операции с плавающей точкой слишком мал для формата назначения, результатом будет положительный или отрицательный нуль.
- Если результат операции с плавающей точкой слишком велик для формата назначения, результатом будет положительная или отрицательная бесконечность.
- Если операция с плавающей точкой является недопустимой, результатом будет NaN.
- Если один или оба операнда операции с плавающей точкой есть NaN, результатом будет NaN.

Операции с плавающей точкой могут выполняться с более высокой точностью, чем тип результата операции. Например, некоторые архитектуры аппаратно реализуют `extended` или `long double` тип с плавающей запятой с более широким диапазоном значений и точностью, чем у типа `double`, и выполняют все операции с плавающей запятой, используя этот более точный тип. Только с дополнительными затратами производительности можно заставить такие аппаратные архитектуры выполнять операции с плавающей точкой с *меньшей* точностью. Чтобы не поплатиться и точностью, и производительностью, C# дает возможность использовать высокоточный тип для всех операций с плавающей точкой. Эта особенность редко приводит к другим сколько-нибудь заметным эффектам, кроме получения более точных результатов. Однако в выражениях вида `x * y / z`, где умножение приводит к появлению результата, выходящего за границы интервала значений `double`, а последующее деление возвращает временный результат в интервал значений `double`, возможность вычислять выражение в формате более высокой точности может привести к получению конечного результата вместо бесконечности.

ДЖОЗЕФ АЛЬБАХАРИ

Значения NaN иногда используются для представления особых значений. В Microsoft's Windows Presentation Foundation `double.NaN` представляет значения, вычисляемые «автоматически». Другой способ представлять такие значения — использовать обнуляемый тип; еще один способ — использовать специальную структуру, которая «обертывает» числовой тип и добавляет другое поле.

4.1.7. Десятичный тип

Десятичный тип (`decimal`) — это 128-разрядный тип данных, использующийся для финансовых вычислений. Десятичный тип может представлять значения в интервале от 1.0×10^{-28} до приблизительно 7.9×10^{28} с 28 или 29 значащими цифрами.

Конечная последовательность значений типа `decimal` существует в форме $(-1)^s \times c \times 10^{-e}$, где s равен 0 или 1, коэффициент c лежит в пределах $0 \le c < 2^{96}$, а степень e лежит в пределах $0 \le e \le 28$. Десятичный тип не поддерживает нулей со знаком, бесконечностей и NaN. Тип `decimal` представлен 96-разрядным целым, масштабированным показателем степени десяти. Для значений `decimal` с абсолютным значением, меньшим `1.0m`, значение определяется с точностью до 28-й десятичной цифры, но не более. Для значений `decimal` с абсолютным значением, большим или равным `1.0m`, значение определяется с точностью до 28 или 29 цифр. В отличие от типов `float` и `double`, десятичные дробные числа, такие как 0.1, могут быть точно представлены в `decimal`. В форматах `float` и `double` такие числа часто являются бесконечными дробями, что делает эти форматы более подверженными ошибкам округления.

ПИТЕР СЕСТОФТ

Стандарт IEEE 754-2008 описывает десятичный тип с плавающей точкой, называемый `decimal128`. Он подобен типу `decimal`, описанному здесь, но упаковывает в те же самые 128 битов намного больше. Он имеет 34 значащие десятичные цифры, интервал значений от 10^{6134} до 10^{6144} и поддерживает NaN. Он был разработан Майком Коулишоу (Mike Cowlishaw) из корпорации IBM. Так как он во всех отношениях расширяет используемый сейчас `decimal`, было бы вполне разумно перейти в будущих версиях C# на стандарт IEEE decimal128.

Если один из операндов бинарной операции типа `decimal`, другой операнд должен быть целого типа или типа `decimal`. Операнд целого типа перед выполнением операции преобразуется к типу `decimal`.

БИЛЛ ВАГНЕР

Нельзя смешивать тип `decimal` и типы с плавающей точкой (`float`, `double`). Такое правило существует, поскольку при смешивании этих типов в процессе вычислений может теряться точность. При совместном использовании типов `decimal` и типов с плавающей точкой необходимо выполнять явное преобразование типа.

Результат операции над значениями типа `decimal` есть результат точного вычисления (с сохранением степени, как определено для каждой операции), округленный до соответствия формату. Результаты округляются до ближайшего представимого значения, и если результат одинаково близок к двум представимым значениям, округление выполняется до значения, которое имеет четную наименьшую значащую цифру (так называемое «банковское округление»). Нулевой результат всегда имеет знак ноль и масштаб 0.

> **ЭРИК ЛИППЕРТ**
>
> Этот метод привлекателен тем, что он обычно приводит к меньшему количеству систематических ошибок, чем методы, которые округляют всегда до большей или до меньшей цифры, когда возникает «вилка» между двумя возможностями.
>
> Как ни странно, несмотря на его название, мало свидетельств тому, что этот метод широко используется в банковском деле.

Если десятичная арифметическая операция приводит к появлению значения, меньшего или равного по абсолютной величине 5×10^{-29}, результатом операции будет ноль. Если десятичная арифметическая операция приводит к появлению значения, слишком большого для десятичного формата, генерируется исключение `System.OverflowException`.

Тип `decimal` имеет более высокую точность, но меньший диапазон значений, чем типы с плавающей точкой. Таким образом, преобразования типов с плавающей точкой к типу `decimal` могут порождать исключения переполнения, а преобразования типа `decimal` к типам с плавающей точкой могут вызвать потерю точности. По этим причинам не существует неявного преобразования между типами с плавающей точкой и `decimal`, и без явного приведения невозможно смешивать в одном и том же выражении операнды с плавающей точкой и операнды типа `decimal`.

> **ЭРИК ЛИППЕРТ**
>
> C# не поддерживает тип данных `Currency`, знакомый пользователям Visual Basic 6 и других языков на основе технологий OLE Automation. Поскольку `decimal` имеет более широкий интервал значений и большую точность, чем `Currency`, все, что может быть сделано с использованием `Currency`, также может быть сделано и с использованием `decimal`.

4.1.8. Булевский тип

Тип `bool` представляет булевские логические значения. Возможные значения типа `bool` есть `true` и `false`.

Стандартных преобразований между булевским и другими типами не существует. В частности, булевский тип отделен от целых типов; булевское значение не может использоваться вместо целочисленного значения, и наоборот.

В языках C и C++ нулевое значение целого типа, типа с плавающей точкой или нулевой указатель могут быть преобразованы к булевскому значению `false`, а ненулевое значение целого типа, типа с плавающей точкой или ненулевой указатель могут быть преобразованы к булевскому значению `true`. В C# такие преобразования выполняются путем явного сравнения целого значения или значения с плавающей точкой с нулем или путем явного сравнения ссылки со значением `null`.

КРИС СЕЛЛЗ

Невозможность преобразовать не-булевские значения к типу `bool` чаще всего беспокоит меня в случае сравнения со значением `null`. Например:

```
object obj = null;
if( obj ) { ... }                  // Правильно в C/C++, ошибка в C#
if( obj != null ) { ... }          // Правильно в C/C++/C#
```

4.1.9. Перечислимые типы

Перечислимый тип (перечисление) — это отдельный тип данных, состоящий из именованных констант. Каждое перечисление имеет базовый тип, он должен быть `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`. Набор значений перечислимого типа такой же, как набор значений базового типа. Значения перечисления не ограничены значениями именованных констант. Перечислимые типы определяются с помощью объявлений (раздел 14.1).

ЭРИК ЛИППЕРТ

Это важное замечание: ничто не может помешать присвоить значение не перечислимого типа переменной этого типа. Не стоит полагаться на язык или среду выполнения в том, что они проверят, что экземпляры перечислимых типов находятся в тех границах, в которых вы ожидаете.

ВЛАДИМИР РЕШЕТНИКОВ

CLR поддерживает `char` в качестве базового типа для перечислений. Если вам случится сослаться на сборку, содержащую такой тип в вашем приложении, компилятор C# не распознает этот тип как перечисление и не позволит, например, преобразовать его к целому типу или из него.

4.1.10. Обнуляемые типы

Обнуляемый тип может представлять все значения своего **базового типа** плюс нулевое значение. Обнуляемый тип записывается как `T?`, где `T` — базовый тип. Такой синтаксис является сокращением записи `System.Nullable<T>`, и две эти формы взаимозаменяемы.

Необнуляемый тип-значение, напротив, есть любой тип-значение, кроме `System.Nullable<T>`, и его краткого обозначения `T?` (для любого `T`), плюс любой параметр-тип, который ограничен необнуляемым типом-значением (то есть любой параметр-тип с ограничением `struct`). Тип `System.Nullable<T>` определяет для `T` ограничение типом-значением (раздел 10.1.5), которое означает, что базовый тип обнуляемого типа должен быть необнуляемым типом-значением. Базовый тип для обнуляемого типа не может быть обнуляемым типом или ссылочным типом. Например, `int??` и `string?` являются недопустимыми типами.

Экземпляр обнуляемого типа `T?` имеет два открытых свойства только для чтения:

- Свойство `HasValue` типа `bool`.
- Свойство `Value` типа `T`.

Экземпляр, для которого `HasValue` имеет значение `true`, называется ненулевым. Ненулевой экземпляр содержит некоторое значение, и `Value` возвращает это значение.

Экземпляр, для которого `HasValue` имеет значение `false`, называется нулевым. Нулевой экземпляр имеет неопределенное значение. Попытка прочитать `Value` нулевого экземпляра приводит к выбрасыванию исключения `System.InvalidOperationException`. Процесс получения доступа к свойству `Value` обнуляемого экземпляра называется *развертыванием* (*unwrapping*).

В дополнение к конструктору по умолчанию каждый обнуляемый тип `T?` имеет открытый конструктор с единственным аргументом типа `T`. Для значения `x` типа `T` вызов конструктора в форме

```
new T?(x)
```

создает ненулевой экземпляр `T?`, для которого свойство `Value` равно `x`. Процесс создания ненулевого экземпляра обнуляемого типа для данного значения называется *обертыванием (wrapping)*.

Существуют неявные преобразования от литерала `null` к `T?` (раздел 6.1.5) и от `T` к `T?` (раздел 6.1.4).

4.2. Ссылочные типы

К ссылочным типам относятся класс, интерфейс, массив и делегат.

```
ссылочный-тип:
   класс
   интерфейс
   массив
   делегат

класс:
   имя-типа
   object
   dynamic
   string
```

```
интерфейс:
   имя-типа

массив:
   не-массив  спецификаторы-размерности

не-массив:
   тип

спецификаторы-размерности:
   спецификатор-размерности
   спецификаторы-размерности   спецификатор-размерности

спецификатор-размерности:
   [ разделители-размерностейopt ]

разделители-размерностей:
   ,
   разделители-размерностей ,

делегат:
   имя-типа
```

Значением ссылочного типа является *экземпляр* типа, также называемый *объектом*. Специальное значение `null` совместимо со всеми ссылочными типами и говорит об отсутствии экземпляра.

4.2.1. Классы

Класс определяет структуру данных, которая содержит элементы данных (константы и поля), функциональные элементы (методы, свойства, события, индексаторы, операции, конструкторы экземпляров, деструкторы и статические конструкторы) и вложенные типы. Классы поддерживают наследование, механизм, с помощью которого производные классы могут расширять и специализировать родительские классы. Экземпляры классов создаются с помощью *выражений-создания-объектов* (раздел 7.6.10.1).

Классы описаны в главе 10.

Некоторые предопределенные классы имеют специальное значение в языке C#, как показано в следующей таблице.

Тип класса	**Описание**
`System.Object`	Первичный родительский класс для всех остальных типов (раздел 4.2.2)
`System.String`	Строковый тип языка C# (раздел 4.2.3)
`System.ValueType`	Родительский класс для всех типов-значений (раздел 4.1.1)
`System.Enum`	Родительский класс для всех перечислений (глава 14)
`System.Array`	Родительский класс для всех массивов (глава 12)
`System.Delegate`	Родительский класс для всех делегатов (глава 15)
`System.Exception`	Родительский класс для всех исключений (глава 16)

4.2.2. Тип object

Класс `object` является корневым классом для всех остальных типов. Каждый тип в C# прямо или косвенно наследуется от класса `object`.

Ключевое слово `object` является псевдонимом предопределенного класса `System.Object`.

4.2.3. Тип dynamic

Тип `dynamic`, подобно типу `object`, может ссылаться на любой объект. Когда операции выполняются с выражениями динамического типа, их разрешение откладывается до выполнения программы. Таким образом, если операцию нельзя законно применить к объектам, на которые ссылаются, во время компиляции ошибка не выдается. Вместо этого во время выполнения программы выбрасывается исключение, если разрешение операции не удалось.

Динамический тип описан в разделе 4.7, а динамическое связывание в разделе 7.2.2.

4.2.4. Тип string

Тип `string` является бесплодным классом, который наследуется прямо от типа `object`. Экземпляры класса `string` представляют собой символьные строки Unicode.

Значения типа `string` могут быть записаны как строковые литералы (раздел 2.4.4.5).

Ключевое слово `string` является псевдонимом предопределенного класса `System.String`.

4.2.5. Интерфейсы

Интерфейс определяет контракт. Класс или структура, которые реализуют интерфейс, должны придерживаться этого контракта. Интерфейс может наследоваться от множества базовых интерфейсов, а класс или структура могут реализовывать множество интерфейсов.

Интерфейсы описаны в главе 13.

4.2.6. Массивы

Массив представляет собой структуру данных, которая содержит ноль или более переменных, доступных через вычисляемые индексы. Переменные, содержащиеся в массиве, называемые элементами массива, относятся к одному и тому же типу, и этот тип называется типом элементов массива.

Массивы описаны в главе 12.

4.2.7. Делегаты

Делегат есть структура данных, которая ссылается на один или более методов. Для методов экземпляров он также ссылается на соответствующие экземпляры объектов.

Аналогом делегата в C или C++ является указатель на функцию, но он может ссылаться только на статические функции, а делегат может ссылаться как на статические методы, так и на методы экземпляра. В последнем случае делегат хранит не только точку входа метода, но также ссылку на экземпляр объекта, для которого вызывается метод.

Делегаты описаны в главе 15.

> **КРИС СЕЛЛЗ**
>
> Хотя в C++ можно ссылаться на экземпляр функции-члена через указатель функции-члена, но это настолько трудно правильно сделать, что с таким же успехом это свойство могло бы быть недопустимым!

4.3. Упаковка и распаковка

Концепция упаковки и распаковки является центральной в системе типов C#. Она наводит мосты между **типами-значениями** и **ссылочными типами**, позволяя любому значению значимого типа конвертироваться к типу `object` и из него. Упаковка и распаковка обеспечивают единство системы типов, в которой значение любого типа может в конечном счете использоваться как объект.

> **ДЖОЗЕФ АЛЬБАХАРИ**
>
> В более ранних версиях, чем C# 2.0, упаковка и распаковка были основными средствами, с помощью которых можно было писать код для универсальной коллекции, например для списка, стека или очереди. После выхода C# 2.0 обобщенные типы обеспечивают альтернативное решение, которое обеспечивает для статических типов большую производительность и безопасность. Упаковка и распаковка, безусловно, требуют некоторых затрат, поскольку они означают копирование значений, работу с косвенными ссылками и выделение памяти в куче.

> **ДЖЕСС ЛИБЕРТИ**
>
> Я бы пошел дальше и сказал бы, что введение обобщенных типов сместило упаковку и распаковку с центрального места к периферии для всех практических целей, и они представляют интерес только для передачи значений значимых типов в качестве параметров `out` и `ref`.

> **КРИСТИАН НЕЙГЕЛ**
> Дополнительные затраты на выполнение, связанные с упаковкой и распаковкой, обычно незначительны, но могут стать огромными, если вы работаете с большими коллекциями. Обобщенные коллекции помогают решить эту проблему.

4.3.1. Упаковка

Упаковка позволяет неявно преобразовывать *тип-значение* в *ссылочный тип*. Существуют следующие виды преобразования:

- Из любого *типа-значения* в тип `object`.
- Из любого *типа-значения* в тип `System.ValueType`.
- Из любого *необнуляемого типа-значения* в любой *интерфейс*, реализованный типом-значением.
- Из любого *обнуляемого типа* в любой *интерфейс*, реализованный базовым типом для *обнуляемого типа*.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> *Обнуляемый тип* **не реализует** интерфейсы своего базового типа; он просто **может преобразовываться** к ним. Это уточнение может быть важным в некоторых контекстах — например, в контексте проверки ограничений обобщенных типов.

- Из любого *перечисления* в тип `System.Enum`.
- Из любого *обнуляемого-типа* с базовым *перечислением* в тип `System.Enum`.

> **БИЛЛ ВАГНЕР**
> Показателен выбор слова «преобразование» для поведения, которое мы видим в приведенных случаях. Вы не интерпретируете некоторую область памяти как относящуюся к другому типу; вы преобразуете ее. То есть вы имеете дело с другой областью памяти, а не смотрите на одну и ту же область как на две переменные разных типов.

Заметим, что неявное преобразование из параметра-типа будет выполняться как упаковка, если во время выполнения оно заканчивается преобразованием типа-значения к ссылочному типу (раздел 6.1.10).

Упаковка значения *необнуляемого-типа-значения* состоит в размещении экземпляра объекта и копировании значения *необнуляемого-типа-значения* в этот экземпляр.

Упаковка значения *обнуляемого-типа* дает нулевую ссылку, если оно имеет значение `null` (`HasValue` имеет значение `false`), в противном случае это результат развертывания (unwrapping) и упаковки базового значения.

Процесс упаковки значения *необнуляемого-типа-значения* лучше всего понять, представив существование обобщенного **упаковочного класса**, который ведет себя как объявленный следующим образом:

```
sealed class Box<T>: System.ValueType
{
   T value;
   public Box(T t) {
      value = t;
   }
}
```

Упаковка значения `v` типа `T` в этом случае состоит в выполнении выражения `new Box<T>(v)` и возвращении значения типа `object` в качестве результата. Таким образом, операторы

```
int i = 123;
object box = i;
```

концептуально соответствуют

```
int i = 123;
object box = new Box<int>(i);
```

Упаковочного класса, подобного приведенному выше `Box<T>`, в действительности не существует, и динамический тип упакованного значения не является в действительности типом класса. На самом деле упакованное значение типа `T` имеет динамический тип `T`, и проверка динамического типа с помощью операции `is` даст просто тип `T`.

Например,

```
int i = 123;
object box = i;
if (box is int) {
   Console.Write("Box содержит int");
}
```

выводит на консоль строку «`Box содержит int`».

Упаковка подразумевает *создание копии* значения, которое упаковывается. Это отличается от преобразования *ссылочного типа* в тип `object`, при котором значение продолжает ссылаться на тот же самый экземпляр и просто считается объектом наименее производного типа `object`. Например, если есть объявление

```
struct Point
{
   public int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

следующие операторы

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.Write(((Point)box).x);
```

выводят на консоль значение 10, поскольку неявная операция упаковки, которая выполняется при присваивании значения `p` объекту `box`, приводит к появлению копии значения `p`. Если бы вместо этого `Point` был объявлен как `class`, выводилось бы значение 20, поскольку `p` и `box` ссылались бы на один и тот же экземпляр.

> **ЭРИК ЛИППЕРТ**
> Такая возможность является только одной из причин, по которым полезно делать структуры неизменяемыми. Если структура не может меняться, то факт, что при упаковке делается копия, не имеет значения: обе копии всегда будут идентичны.

4.3.2. Распаковка

Распаковка позволяет явно преобразовывать *ссылочный-тип* в *тип-значение*. Существуют следующие виды преобразования:

- Из типа `object` в любой *тип-значение*.
- Из типа `System.ValueType` в любой *тип-значение*.
- Из любого *интерфейса* в любой *необнуляемый-тип-значение*, который реализует *интерфейс*.
- Из любого *интерфейса* в любой *обнуляемый-тип*, базовый тип которого реализует *интерфейс*.
- Из любого типа `System.Enum` в любое *перечисление*.
- Из любого типа `System.Enum` в любой *обнуляемый-тип*, базовым для которого является *перечисление*.

> **БИЛЛ ВАГНЕР**
> Так же как и при упаковке, при распаковке выполняется преобразование. Если вы упаковываете структуру, а затем распаковываете ее, в итоге могут получиться три различных места хранения. Почти наверняка у вас не будет трех переменных, занимающих одно и то же место хранения.

Заметим, что явное преобразование к параметру-типу будет выполняться как распаковка, если во время выполнения оно заканчивается преобразованием из ссылочного типа к типу-значению (раздел 6.2.6).

Операция распаковки в *необнуляемый-тип-значение* включает в себя проверку, что экземпляр объекта есть упакованное значение данного *необнуляемого-типа-значения*, а затем копирование значения из экземпляра.

> **ЭРИК ЛИППЕРТ**
> Хотя допустимо преобразовывать не упакованный `int` к не упакованному `double`, недопустимо преобразовывать упакованный `int` к не упакованному `double` — только

> к не упакованному `int`. Такое ограничение существует, поскольку иначе инструкции по распаковке должны были бы «знать» все правила для преобразования типов, обычно выполняемые компилятором. Если вам требуются преобразования такого вида во время выполнения, используйте не распаковку, а класс `Convert`.

Распаковка в *обнуляемый-тип* дает нулевое значение *обнуляемого-типа*, если исходный операнд есть `null`, в противном случае это обернутый (wrapped) результат распаковки экземпляра объекта в тип, базовый для обнуляемого.

Если вернуться к воображаемому упаковочному классу, описанному в предыдущем разделе, распаковка объекта `box` к *типу-значению* `T` состоит в выполнении выражения `((Box<T>)box).value`. Таким образом, операторы

```
object box = 123;
int i = (int)box;
```

концептуально соответствуют

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

Чтобы во время выполнения распаковка в данный *необнуляемый-тип-значение* прошла успешно, значение исходного операнда должно быть ссылкой на упакованное значение этого *необнуляемого-типа-значения*. Если исходный операнд равен `null`, выбрасывается исключение `System.NullReferenceException`. Если исходный операнд является ссылкой на несовместимый объект, выбрасывается `System.InvalidCastException`.

> **ДЖОН СКИТ**
>
> Спецификация C# не гарантирует работоспособность некоторых преобразований распаковки, допустимых в CLI. Например, распаковка из перечисления в его базовый тип и наоборот не разрешена:
>
> ```
> object o = System.DayOfWeek.Sunday;
> int i = (int) o;
> ```

Это преобразование будет успешным в .NET, но не факт, что это будет так же в различных реализациях C#.

Чтобы распаковка в данный обнуляемый тип во время выполнения программы прошла успешно, значение исходного операнда должно быть или `null`, или ссылкой на упакованное значение базового *необнуляемого-типа-значения* для данного *обнуляемого-типа*. Если исходный операнд является ссылкой на несовместимый объект, выбрасывается исключение `System.InvalidCastException`.

> **КРИС СЕЛЛЗ**
>
> Упаковка и распаковка устроены таким образом, что вам почти никогда не требуется о них думать, пока вы не пытаетесь уменьшить объем используемой памяти (в этом случае
>
> *продолжение* ⇨

профилирование ваш друг!). Однако, если вы видите, что значения `out` или `ref` при вызове метода установлены неправильно, подозревайте упаковку.

4.4. Сконструированные типы

Объявление обобщенного типа задает **неограниченный обобщенный тип**, который используется как заготовка для создания многих различных типов с помощью **аргументов-типов**. Аргументы-типы записываются в угловых скобках (`<` и `>`) непосредственно после имени обобщенного типа. Тип, который содержит по крайней мере один аргумент-тип, называется **сконструированным типом**. Сконструированный тип может использоваться в языке в большинстве мест, где может задаваться имя типа. Неограниченный обобщенный тип может использоваться только в *выражениях-typeof* (раздел 7.6.11).

Сконструированные типы также могут использоваться в выражениях как простые имена (раздел 7.6.2) или при доступе к элементу (раздел 7.6.4).

Когда определяется *имя-пространства-имен-или-типа*, рассматриваются только обобщенные типы с правильным числом параметров. Таким образом, можно использовать один и тот же идентификатор для различных типов, если они имеют разное количество параметров типа. Это полезно, когда в одной и той же программе используются обобщенные и необобщенные классы:

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;
    class X
    {
        Queue q1;              // Не обобщенный Widgets.Queue
        Queue<int> q2;         // Обобщенный Widgets.Queue
    }
}
```

Имя-типа может идентифицировать сконструированный тип, даже когда параметры типа не определены явным образом. Такое может быть, если тип вложен в объявление обобщенного класса, и для поиска имен неявно используется тип экземпляра в объявлении, содержащем данный тип (раздел 10.3.8.6):

```
class Outer<T>
{
    public class Inner {...}
    public Inner i;            // Тип i - это Outer<T>.Inner
}
```

В небезопасном коде сконструированный тип не может использоваться как *неуправляемый-тип* (раздел 18.2).

4.4.1. Аргументы-типы

Каждый аргумент в списке аргументов типа — это просто *тип*.

```
список-аргументов-типов:
        <  аргументы-типы  >

аргументы-типы:
аргумент-тип
        аргументы-типы , аргумент-тип

аргумент-тип:
        тип
```

В небезопасном коде (глава 18) *аргумент-тип* не может быть типом указателя. Каждый аргумент-тип должен удовлетворять всем ограничениям соответствующего параметра-типа (раздел 10.1.5).

4.4.2. Открытые и закрытые типы

Все типы можно разделить на **открытые** (open) типы и **закрытые** (closed) типы. Открытый тип есть тип, который включает в себя параметры-типы. Более подробно:

- Параметр-тип определяет открытый тип.
- Массив является открытым типом тогда и только тогда, когда его элементы открытого типа.
- Сконструированный тип является открытым типом тогда и только тогда, когда у него имеется один и более аргументов-типов открытого типа. Сконструированный вложенный тип является открытым типом тогда и только тогда, когда один или более аргументов-типов содержащего его типа (типов) имеют открытый тип.

Закрытым типом является тип, не являющийся открытым.

Во время выполнения программы весь код, содержащийся внутри объявления обобщенного типа, выполняется в контексте закрытого сконструированного типа, который был создан применением аргументов-типов к объявлению обобщенного типа. Каждый параметр-тип внутри обобщенного типа связан с определенным типом времени выполнения. Обработка всех операторов и выражений во время выполнения всегда происходит с закрытыми типами, а открытые типы существуют только во время компиляции.

Каждый закрытый сконструированный тип имеет свой набор статических переменных, который не используется никакими другими закрытыми сконструированными типами. Так как открытых типов во время выполнения программы не существует, не бывает и статических переменных, связанных с открытым типом. Два закрытых сконструированных типа являются одним и тем же типом, если они сконструированы из одного и того же неограниченного обобщенного типа и их соответствующие аргументы-типы имеют один и тот же тип.

4.4.3. Ограниченные и неограниченные типы

Термин **неограниченный (unbound) тип** относится к необобщенному типу или к неограниченному обобщенному типу. Термин **ограниченный (bound) тип** относится к необобщенному или к сконструированному типу.

> **ЭРИК ЛИППЕРТ**
>
> Да, необобщенные типы относятся одновременно и к ограниченным, и к неограниченным.

Неограниченный тип ссылается на сущность, объявленную с помощью объявления типа. Неограниченный обобщенный тип сам по себе типом не является, он не может использоваться в качестве типа переменной, аргумента или возвращаемого значения и в качестве родительского типа. Единственная конструкция, в которой можно использовать неограниченный обобщенный тип, — выражение `typeof` (раздел 7.6.11).

4.4.4. Соблюдение ограничений

Всякий раз при ссылке на сконструированный тип или обобщенный метод используемые аргументы-типы проверяются на соответствие ограничениям параметра-типа, объявленного для обобщенного типа или метода (раздел 10.1.5). Для каждой секции `where` аргумент типа `A`, соответствующий именованному параметру-типу, проверяется на соблюдение следующих ограничений:

- Пусть ограничение является ограничением класса, интерфейса или типа-параметра, и пусть `C` представляет это ограничение с аргументами-типами, замещающими любые используемые в ограничении параметры-типы. Чтобы удовлетворить ограничению, тип `A` должен быть преобразуем в тип `C` одним из следующих способов:
  - Тождественное преобразование (раздел 6.1.1).
  - Неявное преобразование ссылки (раздел 6.1.6).
  - Преобразование упаковки (раздел 6.1.7), при условии, что тип `A` является необнуляемым типом-значением.
  - Неявная ссылка, упаковка или преобразование из параметра-типа `A` в `C`.
- Если ограничение является ограничением ссылочного типа (`class`), тип `A` должен удовлетворять одному из следующих требований:
  - `A` является интерфейсом, классом, делегатом или массивом. Этому ограничению удовлетворяют и ссылочный тип `System.ValueType`, и ссылочный тип `System.Enum`.
  - `A` является параметром-типом, для которого известно, что он имеет ссылочный тип (раздел 10.1.5).
- Если ограничение является ограничением типа-значения (`struct`), тип `A` должен удовлетворять одному из следующих требований:

    - `A` является структурой или перечислением, но не обнуляемым типом. И `System.ValueType`, и `System.Enum` являются ссылочными типами, которые не удовлетворяют этому ограничению.
    - `A` является параметром-типом, имеющим ограничение типа-значения (раздел 10.1.5).
- Если ограничение является ограничением конструктора `new()`, тип `A` не должен быть абстрактным и должен иметь открытый конструктор без параметров. Это требование удовлетворяется, если выполняется один из следующих пунктов:
    - `A` является типом-значением, а все типы-значения имеют открытые конструкторы по умолчанию (раздел 4.1.2).
    - `A` является параметром-типом, имеющим ограничение конструктора (раздел 10.1.5).
    - `A` является параметром-типом, имеющим ограничение типа-значения (раздел 10.1.5).
    - `A` является классом, который содержит явно объявленный открытый конструктор без параметров и не является абстрактным.
    - `A` не является `abstract` и имеет конструктор по умолчанию (раздел 10.11.4).

Если одно или более ограничение параметров-типов для данных аргументов-типов не соблюдается, выдается ошибка компиляции.

Поскольку параметры-типы не наследуются, ограничения тоже не наследуются. В следующем примере нужно определить для параметра-типа `T` класса `D` ограничение, так чтобы `T` удовлетворял ограничению, установленному для родительского класса `B<T>`. Для класса `E`, напротив, не нужно определять ограничение, поскольку `List<T>` реализует `IEnumerable` для любого `T`.

```
class B<T> where T: IEnumerable {...}
class D<T>: B<T> where T: IEnumerable {...}
class E<T>: B<List<T>> {...}
```

4.5. Параметры-типы

Параметр-тип является идентификатором, обозначающим тип-значение или ссылочный тип, с которым связан параметр во время выполнения программы.

параметр-тип:
  идентификатор

Так как параметр-тип может быть инстанцирован значениями многих различных фактических аргументов-типов, параметры-типы имеют немного другие операции и ограничения, чем остальные типы:

- Параметр-тип не может использоваться непосредственно для того, чтобы объявлять родительский класс (раздел 10.2.4) или интерфейс (раздел 13.1.3).
- Правила поиска элементов (member lookup) для параметров-типов зависят от применяемых к ним ограничений, если таковые существуют. Более подробно см. в разделах 6.1.10 и 6.2.6.

- Константу `null` нельзя преобразовать к типу, определяемому параметром-типом, за исключением случая, когда известно, что параметр-тип является ссылочным типом (раздел 6.1.10). Однако вместо этого можно использовать выражение `default` (раздел 7.6.13). Добавим, что тип, определяемый параметром-типом, *можно* сравнивать с `null` с помощью операций `==` и `!=` (раздел 7.10.6), если параметр-тип не имеет ограничения типа-значения.
- Выражение `new` (раздел 7.6.10.6) можно использовать с параметром-типом только если параметр-тип ограничен *ограничением-конструктора* или ограничением типа-значения (раздел 10.1.5).
- Параметр-тип нельзя использовать внутри атрибута.
- Параметр-тип нельзя использовать при доступе к элементу (раздел 7.6.4) или в имени типа (раздел 3.8) для описания статического элемента или вложенного типа.
- В небезопасном коде параметр-тип нельзя использовать как *неуправляемый-тип* (раздел 18.2).

В качестве типа параметр-тип используется исключительно при компиляции. Во время выполнения каждый параметр-тип связан с фактическим типом времени выполнения, как определено набором аргументов-типов в объявлении обобщенного типа. Таким образом, тип переменной, объявленной с параметром-типом, будет во время выполнения закрытым сконструированным типом (раздел 4.4.2). Во время выполнения все операторы и выражения, содержащие параметр-тип, используют фактический тип, определенный аргументом-типом для данного параметра.

4.6. Типы деревьев выражений

Деревья выражений позволяют представлять анонимные функции не как исполняемый код, а как структуры данных. Дерево выражений является значением **типа дерева выражений** в форме `System.Linq.Expressions.Expression<D>`, где `D` — любой делегат. В дальнейшем мы будем пользоваться сокращенным обозначением `Expression<D>`.

Если существует преобразование из анонимной функции к типу делегата `D`, существует также преобразование к типу дерева выражений `Expression<D>`. Преобразование анонимной функции к типу делегата создает делегат, который хранит ссылку на исполняемый код анонимной функции, а преобразование к типу дерева выражений создает представление дерева выражений для анонимной функции.

Деревья выражений являются эффективным представлением анонимных функций в памяти в виде данных и делают структуру анонимной функции явной и прозрачной.

Подобно делегату `D`, дерево выражений `Expression<D>` должно иметь такие же, как у типа `D`, типы параметров и результата.

Следующий пример представляет анонимную функцию как в виде исполняемого кода, так и в виде дерева выражений. Поскольку существу-

ет преобразование к `Func<int,int>`, существует также и преобразование к `Expression<Func<int,int>>`:

```
Func<int,int> del = x => x + 1;                 // Код
Expression<Func<int,int>> exp = x => x + 1;     // Данные
```

В соответствии с этими операциями присваивания делегат `del` ссылается на метод, который возвращает `x + 1`, а дерево выражений `exp` ссылается на структуру данных, которая описывает выражение `x => x + 1`.

Точное определение обобщенного типа `Expression<D>`, а также точные правила для конструирования дерева выражений при преобразовании анонимной функции к типу дерева выражений зависят от реализации.

Важно прояснить две вещи:

- Не все анонимные функции могут быть представлены в виде деревьев выражений. Например, нельзя представить таким образом анонимные функции, содержащие тело операторов, и анонимные функции, содержащие выражения присваивания. В таких случаях преобразование все же существует, но не будет успешно выполнено при компиляции.
- В `Expression<D>` есть метод экземпляра `Compile`, который формирует делегат типа `D`:

```
Func<int,int> del2 = exp.Compile();
```

Вызов этого делегата приводит к выполнению кода, представленного деревом выражений. Таким образом, для данных ранее определений `del` и `del2` эквивалентны, и следующие два оператора работают одинаково:

```
int i1 = del(1);
int i2 = del2(1);
```

После выполнения этого кода `i1` и `i2` будут иметь значение 2.

4.7. Тип dynamic

Тип `dynamic` имеет особое значение в C#. Он призван обеспечить динамическое связывание, которое детально описано в разделе 7.2.2.

Тип `dynamic` идентичен типу `object`, кроме следующих моментов:

- Операции над выражениями типа `dynamic` могут быть динамически связанными (раздел 7.2.2).
- При выведении типов (раздел 7.5.2), если в качестве кандидатов присутствуют и `object`, и `dynamic`, отдается предпочтение типу `dynamic`.

В связи с эквивалентностью этих типов можно утверждать следующее:

- Существует неявное тождественное преобразование между типами `object` и `dynamic`, а также между сконструированными типами, которые будут одинаковыми, если `dynamic` заменить на `object`.
- Явные и неявные преобразования из типа `object` и к нему также применимы к типу `dynamic`.

- Сигнатуры метода, которые не изменяются при замене типа `dynamic` на `object`, считаются одинаковыми сигнатурами.

  Во время выполнения программы тип `dynamic` неотличим от типа `object`. Выражение типа `dynamic` называется *динамическим выражением*.

ЭРИК ЛИППЕРТ

Тип `dynamic` является необычным типом, но важно отметить, что с точки зрения компилятора он *является* типом. В отличие от `var` он может использоваться в большинстве ситуаций, в которых используется тип: как тип результата, параметр-тип, аргумент-тип и т. д.

ПИТЕР СЕСТОФТ

Действительно, `var` является зарезервированным словом и для компилятора типом не является, в то время как `dynamic` является для компилятора типом. Ключевое слово `var` говорит компилятору: «Пожалуйста, выведи тип времени компиляции для этой переменной из ее выражения инициализации». Тип `dynamic`, по существу, говорит компилятору: «Не стоит беспокоиться насчет проверки выражений, в которых появляется эта переменная, в процессе компиляции; система во время выполнения программы все сделает правильно, основываясь на типе времени выполнения *значения* переменной (или выбросит исключение там, где компилятор выдал бы ошибку типа)». Тип `dynamic` не может использоваться как получатель (тип `this`) в методе расширения, как родительский тип или как тип, связанный с обобщенным параметром-типом, но в остальных случаях он может применяться так же, как любой другой тип.

МАРЕК САФАР

Сигнатуры метода рассматриваются как одинаковые при использовании типов `dynamic` и `object`. Это позволяет использовать изящный прием: метод интерфейса, объявленный с типом `object`, может быть непосредственно реализован с использованием типа `dynamic`.

КРИС СЕЛЛЗ

Я впадаю в задумчивость, когда в каком-либо языке следующая строка символов допустима и имеет смысл:

```
class Foo {
   public static dynamic DoFoo() {...}
}
```

Конечно, это означает, что метод `DoFoo` является методом типа (в отличие от метода экземпляра) и что тип возвращаемого значения неизвестен до выполнения программы, но в глаза бросается соседство слов `static` и `dynamic`, и такая странность несколько нервирует.

Глава 5
Переменные

Переменные представляют собой ячейки хранения. Каждая переменная имеет тип, который определяет, какие значения могут в ней храниться. C# является языком, поддерживающим безопасность типов, и компилятор C# гарантирует, что значения, содержащиеся в переменных, всегда имеют соответствующий тип. Значение переменной может быть изменено с помощью присваивания или с использованием операций `++` и `--`.

Переменная должна быть **явно присвоена** (раздел 5.3), прежде чем ее значение можно будет использовать.

Как описано в следующих разделах, переменные бывают **инициализированные** и **неинициализированные**. Инициализированная переменная имеет полностью определенное начальное значение и всегда рассматривается как явно присвоенная. Неинициализированная переменная начального значения не имеет. Чтобы такая переменная стала явно присвоенной в некоторой точке программы, присваивание значения этой переменной должно быть выполнено на каждом возможном пути к этой точке.

5.1. Категории переменных

В C# семь категорий переменных: статические переменные, переменные экземпляра, элементы массива, параметры-значения, параметры-ссылки, выходные параметры и локальные переменные. В следующих разделах описана каждая из этих категорий.

В примере

```
class A
{
    public static int x;
    int y;
    void F(int[] v, int a, ref int b, out int c)
    {
        int i = 1;
        c = a + b++;
    }
}
```

`x` — статическая переменная, `y` — переменная экземпляра, `v[0]` — элемент массива, `a` — параметр-значение, `b` —параметр-ссылка, `c` —выходной параметр, `i` — локальная переменная.

ДЖЕСС ЛИБЕРТИ

К несчастью, неизбежно, что `x` первым делом используется в качестве имени переменной во всех наших книгах. Имена переменных должны говорить сами за себя, за исключением счетчиков цикла; никогда не следует давать имена, состоящие из одной буквы. Я бы предпочел следующую запись приведенного выше примера:

```
class ASimpleExampleClass
{
    public static int staticMember;
    int memberVariable;
    void ExampleFunction(
        int[] arrayOfIntsParam,
        int simpleParam,
        ref int refParam,
        out int OutParam)
    {
        int tempVariable = 1;
        outParam = simpleParam + refParam
    }
}
```

Хотя такая схема именования может показаться громоздкой для простого примера, зато в ней нет никаких двусмысленностей — сразу понятно, что происходит и почему — и все последующие объяснения становятся ненужными.

5.1.1. Статические переменные

Поля, объявленные с модификатором `static`, называются статическими переменными. Статическая переменная начинает существование перед выполнением статического конструктора (раздел 10.12) содержащего ее типа, а перестает существовать, когда перестает существовать связанный с ней домен приложения.

Начальным значением статической переменной является значение по умолчанию (раздел 5.2) для переменной этого типа.

С позиций проверки явного присваивания считается, что значение статической переменной присвоено изначально.

5.1.2. Переменные экземпляра

Поля, объявленные без модификатора `static`, называются **переменными экземпляра**.

5.1.2.1. Переменные экземпляра в классах

Переменные экземпляра класса начинают существование при создании нового экземпляра класса и прекращают существовать, когда отсутствуют ссылки на этот экземпляр и выполнен деструктор экземпляра (если таковой существовал).

Начальным значением переменной экземпляра класса является значение по умолчанию (раздел 5.2) для переменной этого типа.

С позиций проверки явного присваивания считается, что значение статической переменной экземпляра присвоено изначально.

5.1.2.2. Переменные экземпляра в структурах

Переменная экземпляра структуры имеет такое же время жизни, как структурная переменная, которой она принадлежит. Иными словами, когда переменная структурного типа начинает или прекращает существовать, то же самое происходит и с переменной экземпляра структуры.

Наличие или отсутствие начального значения переменной экземпляра структуры соответствует наличию или отсутствию начального значения соответствующей структурной переменной. Иными словами, если структурная переменная инициализирована, то инициализированы и составляющие ее переменные, и наоборот.

> **БИЛЛ ВАГНЕР**
>
> Переменная экземпляра ссылочного типа в структуре может не подлежать сборке мусора, когда содержащая ее структура прекращает существовать. Если объект достижим по другому пути, он все еще продолжает «жить», хотя экземпляр структуры уже не существует. То же самое верно для элементов массива.

5.1.3. Элементы массива

Элементы массива начинают существование при создании массива и прекращают существование, когда нет ссылок на экземпляр массива.

Начальным значением каждого из элементов массива является значение по умолчанию (раздел 5.2) для элементов этого типа.

С позиций проверки явного присваивания считается, что значение элементу массива присвоено изначально.

5.1.4. Параметры-значения

Параметры, объявленные без модификаторов `ref` или `out`, являются **параметрами-значениями**.

Параметры-значения начинают существование после вызова функционального элемента (метода, конструктора экземпляра, кода доступа или операции) или анонимной функции, к которой относится параметр, и инициализируются значением аргумента, переданным во время вызова. Параметр-значение обычно прекращает существовать при возврате из функционального элемента или анонимной функции. Однако, если параметр-значение «захвачен» анонимной функцией, (раздел 7.15), его время жизни увеличится — по меньшей мере до того момента, когда делегат

или дерево выражений, созданные из этой анонимной функции, будут подлежать сборке мусора.

С позиций проверки явного присваивания считается, что значение параметру-значению присвоено изначально.

5.1.5. Параметры-ссылки

Параметры, объявленные с модификатором `ref`, являются **ссылочными параметрами**.

Для параметра-ссылки не создается новой ячейки хранения. Напротив, параметр-ссылка представляет ту же ячейку хранения, что у переменной, переданной в качестве аргумента функционального элемента или вызова анонимной функции. Таким образом, значение параметра-ссылки всегда такое же, как у нижележащей переменной.

К параметрам-ссылкам применимы перечисленные далее правила явного присваивания. Отметим, что отличающиеся от них правила для выходных параметров описаны в разделе 5.1.6.

- Переменная должна быть явно присвоена (раздел 5.3), прежде чем ее можно будет передать как параметр-ссылку функциональному элементу или вызову делегата.
- Внутри функционального элемента или анонимной функции параметр-ссылка считается инициализированным.

Внутри метода экземпляра или кода доступа к экземпляру структурного типа ключевое слово `this` ведет себя в точности как параметр-ссылка структурного типа (раздел 7.6.7).

ЭРИК ЛИППЕРТ

Менее формально, различие между параметром-ссылкой и выходным параметром состоит в том, что параметр-ссылка представляет собой «входной и выходной» параметр. Он должен быть инициализирован при вызове метода, и метод может изменять его содержимое. Выходной параметр используется для формирования результата; метод должен наполнить его содержимым и не смотрит на это содержимое вплоть до того момента, когда это сделано.

ДЖОН СКИТ

Даже выходные параметры не обязательно используются *исключительно* для вывода. В частности, после того как метод присваивает значение выходному параметру, он затем может его прочитать. Однако, поскольку параметр будет разделять ячейку хранения с другой переменной, нет гарантий, что значение не изменится снова после того, как ему будет выполнено присваивание в методе. Такой подход препятствует ковариантному использованию выходных переменных.

ЭРИК ЛИППЕРТ

Некоторые другие языки программирования поддерживают другой вид передачи параметров: ссылка «только для ввода» (input only). То есть ссылка на переменную передается методу, но, в отличие от ссылочных параметров в C#, метод может только читать из нее, а записывать в нее не может. Такой подход полезен для эффективной передачи данных большого объема; передача ссылки на переменную может выполняться лучше, чем передача копии значения, если значение велико. C# не поддерживает этот вид передачи по ссылке.

5.1.6. Выходные параметры

Параметр, объявленный с модификатором `out`, является **выходным параметром**.

Для выходного параметра не создается новой ячейки хранения. Вместо этого выходной параметр представляет ту же ячейку хранения, что у переменной, переданной в качестве аргумента функциональному элементу или вызову делегата. Таким образом, значение выходного параметра всегда такое же, как у нижележащей переменной.

К выходным параметрам применимы перечисленные далее правила явного присваивания. Отметим, что отличающиеся от них правила для ссылочных параметров описаны в разделе 5.1.5.

- Переменная не обязательно должна быть явно присвоена, прежде чем ее можно будет передать как выходной параметр функциональному элементу или вызову делегата.
- При нормальном завершении функционального элемента или вызова делегата считается, что каждой переменной, переданной как выходной параметр, на данном пути выполнения присвоено значение.
- Внутри функционального элемента или анонимной функции выходной параметр считается неинициализированным.
- Каждый выходной параметр функционального элемента или анонимной функции должен быть явно присвоен (раздел 5.3) до нормального завершения функционального элемента или анонимной функции.

Внутри конструктора экземпляра структурного типа ключевое слово `this` ведет себя как выходной параметр структурного типа (раздел 7.6.7).

КРИСТИАН НЕЙГЕЛ

Для возврата из функции множества значений вместо использования выходных параметров можно использовать кортежный тип (tuple type). Этот тип появился в .NET 4.

5.1.7. Локальные переменные

Локальная переменная объявляется либо в *объявлении-локальной-переменной*, которое может встретиться в *блоке*, в *операторе-for*, в *операторе-switch* и в *операторе-using*, либо в *операторе-foreach*, либо в *catch-блоке оператора-try*.

Время жизни локальной переменной есть та часть времени выполнения программы, в течение которой гарантировано хранение этой переменной. Это время простирается по меньшей мере от входа в *блок*, *оператор-for*, *оператор-switch*, *оператор-using*, *оператор-foreach* или в *catch-блок оператора-try*, с которым связана эта переменная, и до тех пор пока выполнение упомянутого блока или оператора не завершится тем или иным способом. (Вход во вложенный блок или вызов метода откладывает, но не заканчивает выполнение текущего блока или оператора, перечисленных выше.) Если локальная переменная «захватывается» анонимной функцией (раздел 7.15.5.1), ее время жизни увеличивается по меньшей мере до того момента, как делегат или дерево выражений, созданные из анонимной функции, вместе с другими объектами, которые ссылаются на «захваченную» переменную, становятся подлежащими сборке мусора.

> **БИЛЛ ВАГНЕР**
>
> Последнее замечание имеет важные следствия для возможных затрат, связанных с локальными переменными, «захваченными» анонимными функциями. Время жизни таких переменных может быть намного больше, а это означает, что другие объекты, на которые ссылаются эти переменные, также будут существовать намного дольше.

Если вход в родительский *блок*, *оператор-for*, *оператор-switch*, *оператор-using*, *оператор-foreach* или в *catch-блок оператора-try* выполняется рекурсивно, каждый раз создается новый экземпляр локальной переменной и *инициализатор-локальной-переменной*, если таковой существует, вычисляется каждый раз.

Локальная переменная, описанная с помощью *объявления-локальной-переменной*, не инициализируется автоматически и не имеет значения по умолчанию. С позиций проверки явного присваивания считается, что локальная переменная, описанная с помощью *объявления-локальной-переменной*, изначально не инициализирована. *Объявление-локальной-переменной* может включать в себя *инициализатор-локальной-переменной*; в этом случае переменная считается явно присвоенной только после выражения инициализации (раздел 5.3.3.4).

> **ЭРИК ЛИППЕРТ**
>
> Может показаться, что требование явного присваивания для локальных переменных вместо автоматического присваивания им значения по умолчанию вызвано соображениями выигрыша в производительности: ведь компилятор не генерирует код, который без необходимости присваивает ячейке хранения значение по умолчанию. На самом деле это не было мотивом для введения такого свойства. Фактически CLR инициализирует локальные переменные их значениями по умолчанию, причем очень быстро. Мотивирующим фактором для проверки явного присваивания было желание убрать причину многих ошибок. C# не делает предположений о том, что вы опирались на начальное значение, присвоенное автоматически, и таким образом язык не маскирует ваши ошибки; напротив, он требует, чтобы локальные переменные были перед использованием инициализированы явным образом.

Если внутри области видимости локальной переменной, описанной с помощью *объявления-локальной-переменной*, обращаться к ней из позиции, предшествующей *описателю-локальной-переменной*, будет выдана ошибка компиляции. Если локальная переменная объявлена неявно (раздел 8.5.1), то ссылка на локальную переменную внутри *описателя-локальной-переменной* также является ошибкой.

Локальная переменная, описанная в *операторе-foreach* или в *catch-блоке*, считается явно присвоенной во всей области видимости.

Действительное время жизни локальной переменной зависит от реализации. Например, компилятор может статически определить, что локальная переменная, описанная в блоке, используется только в малой части этого блока. Используя анализ, компилятор может сгенерировать код, который приведет к тому, что ячейка хранения переменной будет иметь меньшее время жизни, чем содержащий ее блок.

Память, которая выделяется для хранения локальной ссылочной переменной, используется независимо от времени жизни этой локальной ссылочной переменной (раздел 3.9).

КРИС СЕЛЛЗ

В C требовалось, чтобы переменные были объявлены в начале области видимости:

```
void F() {
    int x = ...;
    ...
    Foo(x);      // Что еще за x?
    ...
}
```

В C#, вообще говоря, считается плохой практикой объявлять переменные вдали от того места, где они будут использоваться. Предпочтительнее следующая форма записи:

```
void F() {
    ...
    int x = ...;
    Foo(x); // Да-да, я вижу x...
    ...
}
```

Принимая во внимание этот принцип, известный как «локализация ссылок» (locality of reference), можно сделать программу более удобной для чтения и понимания.

5.2. Значение по умолчанию

Следующие категории переменных автоматически инициализируются их значениями по умолчанию:

- Статические переменные.
- Переменные экземпляров классов.
- Элементы массива.

Значение по умолчанию зависит от типа переменной и определяется следующим образом:

- Для переменной *типа-значения* значение по умолчанию совпадает со значением, вычисленным конструктором по умолчанию этого *типа-значения* (раздел 4.1.2).
- Для переменной ссылочного типа значением по умолчанию является `null`.

Инициализация значениями по умолчанию обычно выполняется модулем распределения памяти или сборщиком мусора, инициализирующими память обнулением всех битов до ее распределения. По этой причине удобно представлять нулевую ссылку с помощью комбинации всех битовых нулей.

5.3. Явное присваивание

Переменная считается **явно присвоенной** в некоторой точке исполняемого кода функционального элемента, если компилятор, используя статический анализ потока данных (раздел 5.3.3), может удостовериться, что переменная была автоматически инициализирована или ей было присвоено по меньшей мере одно значение. Говоря неформально, правила явной инициализации следующие:

- Инициализированная переменная (раздел 5.3.1) всегда считается явно присвоенной.
- Неинициализированная переменная (раздел 5.3.2) считается явно присвоенной в данном месте программы, если все возможные пути выполнения, ведущие к этому месту, содержат по меньшей мере одну из следующих операций:
    - Простое присваивание значения (раздел 7.17.1), в котором переменная является левым операндом.
    - Выражение вызова (раздел 7.6.5) или выражение создания объекта (раздел 7.6.10.1), которое передает переменную в качестве выходного параметра.
    - Для локальной переменной — ее объявление (раздел 8.5.1), содержащее инициализатор переменной.

Формальная спецификация, лежащая в основе только что приведенных неформальных правил, описана в разделах 5.3.1—5.3.3.

Состояния явного присваивания для полей переменных структурного типа отслеживаются как индивидуально, так и совместно. В добавление к приведенным выше правилам, следующие правила применяются к структурам и их полям:

- Поле структуры считается явно присвоенным, если содержащая ее переменная структурного типа является явно присвоенной.
- Переменная структурного типа считается явно присвоенной, если каждое из ее полей является явно присвоенным.

Явное присваивание требуется в следующих контекстах:

- Переменная должна быть явно присвоенной в каждом месте, где используется ее значение. Это позволяет быть уверенным в том, что неопределенные значения

никогда не появятся. Наличие переменной в выражении подразумевает получение значения переменной, за исключением следующих случаев:

- Переменная является левым операндом операции простого присваивания.
- Переменная передается в качестве выходного параметра.
- Переменная является переменной структурного типа и является левым операндом операции доступа к элементу.

- Переменная должна быть явно присвоена в каждом месте, где она передается как параметр-ссылка. Это дает уверенность, что вызванные функциональные элементы могут рассматривать ссылочный параметр как инициализированный.
- Все выходные параметры функционального элемента должны быть явно присвоены в каждом месте, где выполняется возврат из него (с помощью оператора `return` или при достижении конца тела функционального элемента). Это гарантирует, что функциональные элементы не возвратят неопределенных значений выходных параметров, что позволяет компилятору считать вызов функционального элемента, который берет переменную в качестве выходного параметра, эквивалентным присваиванию значения переменной.
- Переменная `this` конструктора экземпляра структурного типа должна быть явно присвоена в каждом месте, где выполняется возврат из конструктора экземпляра.

5.3.1. Инициализированные переменные

Следующие категории переменных считаются инициализированными:

- Статические переменные.
- Нестатические поля экземпляров классов.
- Нестатические поля экземпляров инициализированных структурных переменных.
- Элементы массивов.
- Параметры-значения.
- Параметры-ссылки.
- Переменные, объявленные в блоке `catch` и операторе `foreach`.

5.3.2. Неинициализированные переменные

Следующие категории переменных считаются неинициализированными:

- Нестатические поля экземпляров неинициализированных структурных переменных.
- Выходные параметры, в том числе переменная `this` конструктора экземпляра структуры.
- Локальные переменные, за исключением объявленных в блоке `catch` и операторе `foreach`.

5.3.3. Точные правила для определения явного присваивания

Чтобы определить, что каждая используемая переменная является явно присвоенной, компилятор должен выполнить алгоритм, подобный описанному в данном разделе.

Компилятор обрабатывает тело каждого функционального элемента, который имеет одну или несколько неинициализированных переменных. Для каждой такой переменной `v` компилятор определяет **состояние явного присваивания** (*definite assignment state*, далее — *DAS*) в каждой из следующих точек функционального элемента:

- В начале каждого оператора.
- В конечной точке (раздел 8.1) каждого оператора.
- В каждой дуге передачи управления другому оператору или в конечной точке оператора.
- В начале каждого выражения.
- В конце каждого выражения.

DAS для `v` может быть одним из следующих:

- Явно присвоена. Это отражает, что во всех возможных путях к данной точке переменной `v` присвоено значение.
- Не является явно присвоенной. Для состояния переменной в конце выражения типа `bool` состояние переменной, которая не является явно присвоенной, может (но не обязательно) переходить в одно из следующих подсостояний:
    - Явно присвоенная после выражения `true`. Состояние отражает, что `v` является явно присвоенной, если результат булевского выражения будет `true`, но не обязательно будет присвоенной, если результат булевского выражения будет `false`.
    - Явно присвоенная после выражения `false`. Состояние отражает, что `v` является явно присвоенной, если результат булевского выражения будет `false`, но не обязательно будет присвоенной, если результат булевского выражения будет `true`.

Следующие правила определяют, как состояние переменной `v` определяется в каждом месте программы.

5.3.3.1. Общие правила для операторов

- `v` не является явно присвоенной в начале тела функционального элемента.
- `v` является явно присвоенной в начале любого недоступного оператора.

> **ЭРИК ЛИППЕРТ**
>
> Это правило может поразить вас своей бесполезностью. Почему каждую переменную внутри недоступного оператора нужно считать явно присвоенной? Некоторые причины мы сейчас обсудим.

> Если оператор недоступен, он не будет выполнен. Если он не будет выполнен, он не использует значение неинициализированной переменной. Таким образом, здесь нет проблемы; компилятор ищет потенциальные проблемы, но здесь их нет.
> Кроме того, недоступные операторы по большей части являются ошибкой. Как только компилятор обнаружит одну ошибку, велик шанс, что вокруг нее возникнет целая куча связанных с ней ошибок. Чем сообщать об огромном количестве связанных между собой проблем, гораздо лучше сообщить только об одной проблеме и дать возможность пользователю обратить на нее внимание.

- DAS для `v` в начале любого оператора определяется проверкой состояния явного присваивания `v` во всех путях, которые ведут к началу данного оператора. В том и только в том случае, если `v` является явно присвоенной во всех таких путях, `v` будет являться явно присвоенной в начале оператора. Набор возможных путей определяется так же, как для проверки доступности оператора (раздел 8.1).
- DAS для `v` в конце блока или операторов `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using` и `switch` определяется проверкой состояния явного присваивания `v` во всех путях, которые ведут к конечной точке оператора. В том и только в том случае, если `v` является явно присвоенной во всех таких путях, `v` будет являться явно присвоенной в конечной точке оператора. В противном случае `v` не является явно присвоенной в конечной точке оператора. Набор возможных путей определяется так же, как для проверки доступности оператора (раздел 8.1).

5.3.3.2. Блок, операторы checked и unchecked

DAS для `v` при передаче управления на первый оператор в списке операторов блока (или на конечную точку блока, если список операторов пуст) такое же, как DAS для `v` перед блоком или операторами `checked` и `unchecked`.

5.3.3.3. Оператор-выражение

Для оператора-выражения *stmt*, который включает в себя выражение *expr*:

- `v` имеет такое же DAS в начале *expr*, как в начале *stmt*.
- Если `v` является явно присвоенной в конце *expr*, она является явно присвоенной в конечной точке *stmt*.

5.3.3.4. Операторы объявления

- Если оператор объявления не содержит инициализаторов, `v` имеет такое же DAS в конечной точке *stmt*, как в начале *stmt*.
- Если *stmt* есть оператор объявления с инициализаторами, то DAS для `v` определяется так, как если бы *stmt* был списком операторов с одним оператором присваивания для каждого объявления с инициализатором (в порядке следования объявлений).

5.3.3.5. Условный оператор

Для оператора `if` оператор *stmt* в форме:

```
if ( expr ) then-stmt else else-stmt
```

- `v` имеет такое же DAS в начале *expr*, как в начале *stmt*.
- Если `v` является явно присвоенной в конце *expr*, то она явно присвоена на пути к *then-stmt* и на пути либо к *else-stmt*, либо, если отсутствует ветвь *else*, к конечной точке *stmt*.
- Если `v` имеет состояние «явно присвоенной после выражения `true`» в конце *expr*, то она явно присвоена на пути к *then-stmt* и не является явно присвоенной на пути либо к *else-stmt*, либо, если отсутствует ветвь *else*, к конечной точке *stmt*.
- Если `v` имеет состояние «явно присвоенной после выражения `false`» в конце *expr*, то она явно присвоена на пути к *else-stmt* и не является явно присвоенной на пути к *then-stmt*. Она явно присвоена в конечной точке *stmt* в том и только в том случае, если она явно присвоена в конечной точке *then-stmt*.
- Иначе `v` не считается явно присвоенной на пути к *then-stmt* или к *else-stmt* или, если отсутствует ветвь *else*, к конечной точке *stmt*.

5.3.3.6. Оператор выбора

В операторе `switch` оператор *stmt* с управляющим выражением *expr*:

- DAS для `v` в начале *expr* такое же, как состояние в начале *stmt*.
- DAS для `v` на пути к списку операторов достижимого блока `switch` такое же, как DAS для `v` в конце *expr*.

ЭРИК ЛИППЕРТ

К сожалению, в некоторых случаях, когда в операторе `switch` перечислены все возможные значения для данного типа (таковы `bool`, `byte` и соответствующие им обнуляемые типы), проверка явного присваивания не принимает это во внимание и считает, что переменная не является явно присвоенной в конце `switch`. Например:

```
int x;
bool b = B();
switch(b) {
case true : x = 1; break;
case false: x = 2; break;
}
Console.WriteLine(x);         // Ошибка: x не является явно присвоенной
```

В таких редких случаях вы всегда можете записать избыточную ветвь по умолчанию `default`, чтобы удовлетворить требования контроля присваивания.

5.3.3.7. Оператор цикла while

Для оператора `while` *stmt* в форме:

```
while ( expr ) while-body
```

- `v` имеет такое же DAS в начале *expr*, как в начале *stmt*.
- Если `v` является явно присвоенной в конце *expr*, то она явно присвоена на пути к *телу-while* и к конечной точке *stmt*.
- Если `v` находится в состоянии «явно присвоенной после выражения `true`» в конце *expr*, то она явно присвоена на пути к телу оператора `while`, но не является явно присвоенной в конечной точке оператора *stmt*.
- Если `v` находится в состоянии «явно присвоенной после выражения `false`» в конце *expr*, то она явно присвоена при передаче управления на конец *stmt*, но не является явно присвоенной на пути к *телу-while*.

5.3.3.8. Оператор цикла do

Для оператора `do` *stmt* в форме:

```
do do-body while ( expr ) ;
```

- `v` имеет такое же DAS на пути от начала *stmt* к *телу-do*, как в начале *stmt*.
- `v` имеет такое же DAS в начале *expr*, как в конечной точке *тела-do*.
- Если `v` является явно присвоенной в конце *expr*, то она явно присвоена на пути к конечной точке *stmt*.
- Если `v` имеет состояние «явно присвоенной после выражения `false`» в конце *expr*, то она является явно присвоенной на пути к конечной точке *stmt*.

5.3.3.9. Оператор цикла for

Проверка DAS переменных для оператора `for` в форме:

```
for ( инициализатор-for ; условие-for ; итератор-for ) вложенный-оператор
```

производится так, как будто оператор записан в форме:

```
{
   инициализатор-for ;
   while (условие-for ) {
         вложенный-оператор ;
         итератор-for ;
   }
}
```

Если *условие-for* в операторе `for` опущено, определение DAS производится так, как будто *условие-for* заменено на `true`.

5.3.3.10. Операторы перехода break, continue и goto

DAS для `v` при передаче управления, выполняемой операторами `break`, `continue`, или `goto` такое же, как DAS для `v` в начале оператора.

5.3.3.11. Оператор throw

Для оператора *stmt* в форме:

```
throw expr ;
```

DAS для `v` в начале *expr* такое же, как DAS для `v` в начале оператора *stmt*.

5.3.3.12. Оператор return

Для оператора *stmt* в форме :

```
return expr ;
```

- DAS для v в начале *expr* такое же, как DAS для v в начале оператора *stmt*.
- Если v является выходным параметром, то она должна быть явно присвоена одним из следующих способов:
  - После *expr*.
  - В конце завершающего блока `finally` в виде `try-finally` или `try-catch-finally`, который содержит оператор `return`.

Для оператора *stmt* в форме:

```
return ;
```

- Если v является выходным параметром, то она должна быть явно присвоена одним из следующих способов:
  - До оператора *stmt*.
  - В конце завершающего блока `finally` в виде `try-finally` или `try-catch-finally`, который содержит оператор `return`.

5.3.3.13. Оператор try-catch

> **БИЛЛ ВАГНЕР**
>
> Начиная с этого места вы можете видеть, как несколько неструктурных операторов (`throw`, `catch`, `goto`, `finally`) могут усложнить как анализ, проводимый компилятором, так и ваше собственное понимание. Будьте осторожны с использованием этих операторов в вашей обычной логике. Оператор `try/finally` является упрощенным особым случаем, но, в общем случае, он может сильно ухудшить читабельность программы.

Для оператора *stmt* в форме:

```
try блок-try
catch(...) блок-catch-1
...
catch(...) блок-catch-n
```

- DAS для v в начале *блока-try* такое же, как DAS для v в начале оператора *stmt*.
- DAS для v в начале *блока-catch-i* (для любого *i*) такое же, как DAS для v в начале *stmt*.
- В конечной точке *stmt* переменная v является явно присвоенной в том и только в том случае, если v явно присвоена в конечной точке *блока-try* и каждого *блока-catch-i* (для любого *i* от 1 до *n*).

5.3.3.14. Оператор try-finally

Для оператора `try` *stmt* в форме:

```
try блок-try finally блок-finally
```

- DAS для `v` в начале *блока-try* такое же, как DAS для `v` в начале *stmt*.
- DAS для `v` в начале *блока-finally* такое же, как DAS для `v` в начале *stmt*.
- Переменная `v` в конечной точке *stmt* является явно присвоенной в том и только в том случае, если по меньшей мере одно из следующих утверждений верно:
  - `v` является явно присвоенной в конечной точке блока *блока-try*.
  - `v` является явно присвоенной в конечной точке блока *блока-finally*.

Если выполнена передача управления (например, оператор `goto`) изнутри *блока-try* наружу *блока-try*, то `v` считается явно присвоенной при этой передаче управления, если `v` явно присвоена в конечной точке блока *блока-finally*. (Здесь не «только если»: если `v` явно присвоена при этой передаче управления другим образом, она также будет считаться явно присвоенной.)

5.3.3.15. Оператор try-catch-finally

Анализ DAS для оператора `try-catch-finally` в форме:

```
try блок-try
catch(...) блок-catch-1
...
catch(...) блок-catch-n
finally блок-finally
```

выполняется так, как если бы оператор был оператором `try-finally`, содержащим оператор `try-catch`:

```
try {
  try блок-try
  catch(...) блок-catch-1
  ...
  catch(...) блок-catch-n
}
finally блок-finally
```

Следующий пример демонстрирует, как различные блоки оператора `try` (раздел 8.10) влияют на DAS.

```
class A
{
    static void F()
    {
        int i, j;
        try
        {
            goto LABEL;
            // Ни i, ни j не являются явно присвоенными
            i = 1;
            // i явно присвоена
        }
        catch
        {
            // Ни i, ни j не являются явно присвоенными
            i = 3;
            // i явно присвоена
}
```

продолжение ⇨

```
        finally
        {
            // Ни i, ни j не являются явно присвоенными
            j = 5;
            // j явно присвоена
        }
    // i и j явно присвоены
    LABEL: ;
        // j явно присвоена
    }
}
```

5.3.3.16. Оператор foreach

Для оператора `foreach` *stmt* в форме:

foreach (*тип идентификатор* **in** *expr* **)** *вложенный-оператор*

- DAS для `v` в начале *expr* такое же, как DAS для `v` в начале *stmt*.
- DAS для `v` на пути к *вложенному-оператору* или к конечной точке *stmt* такое же, как DAS для `v` в конце *expr*.

5.3.3.17. Оператор using

Для оператора `using` stmt в форме:

using (*получение-ресурса* **)** *вложенный-оператор*

- DAS для `v` в начале *получения-ресурса* такое же, как состояние `v` в начале *stmt*.
- DAS для `v` на пути к *вложенному-оператору* такое же, как DAS для `v` в конце *получения-ресурса*.

5.3.3.18. Оператор lock

Для оператора `lock` *stmt* в форме:

lock (*expr* **)** *вложенный-оператор*

- DAS для `v` в начале *expr* такое же, как DAS для `v` в начале *stmt*.
- DAS для `v` на пути к *вложенному-оператору* такое же, как DAS для `v` в конце *expr*.

5.3.3.19. Оператор yield

Для оператора `yield return` *stmt* в форме:

yield return *expr* **;**

- DAS для `v` в начале *expr* такое же, как DAS для `v` в начале *stmt*.
- DAS для `v` в конце *stmt* такое же, как DAS для `v` в конце *expr*.

  Оператор `yield break` на DAS не влияет.

5.3.3.20. Общие правила для простых выражений

Приведенные ниже правила применимы к следующим видам выражений: константы (раздел 7.6.1.), простые имена (раздел 7.6.2), выражения доступа к элементу (раздел 7.6.4), неиндексированные выражения доступа к базовому классу (раздел 7.6.8), выражения `typeof` (раздел 7.6.11) и выражения значений по умолчанию (раздел 7.6.13).

- DAS для `v` в конце такого выражения такое же, как DAS для `v` в начале выражения.

5.3.3.21. Общие правила для выражений с вложенными выражениями

Приведенные ниже правила применимы к следующим видам выражений: выражения, заключенные в круглые скобки (раздел 7.6.3), выражения доступа к элементу (раздел 7.6.6), индексированные выражения доступа к базовому классу (раздел 7.6), выражения инкремента и декремента (разделы 7.6.9, 7.7.5), выражения преобразования типа (раздел 7.7.6), унарные выражения `+`, `-`, `~`, `*`, бинарные выражения `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` (разделы 7.8—7.11), выражения сложного присваивания (раздел 7.17.2); выражения `checked` и `unchecked` (раздел 7.6.12); а также выражения создания массивов и делегатов (раздел 7.6.10).

Каждое из таких выражений имеет одно или более подвыражений, которые, безусловно, вычисляются в фиксированном порядке. Например, бинарная операция % вычисляет левую часть операции, затем правую часть. Операция индексирования вычисляет индексированное выражение, а затем вычисляет выражения для каждого индекса слева направо. Для выражения *expr* имеются подвыражения $expr_1$, $expr_2$, ..., $expr_n$, вычисляемые в следующем порядке:

- DAS для `v` в начале $expr_1$ такое же, как DAS в начале *expr*.
- DAS для `v` в начале $expr_i$ такое же, как DAS в конце $expr_{i-1}$.
- DAS для `v` в конце *expr* такое же, как DAS в конце $expr_n$.

5.3.3.22. Выражения вызова и выражения создания объекта

Для выражения вызова *expr* в форме

```
первичное-выражение ( arg1 , arg2 , ... , argn )
```

или для выражения создания объекта в форме

```
new тип ( arg1 , arg2 , ... , argn )
```

- Для выражения вызова DAS для `v` перед *первичным-выражением* такое же, как состояние `v` перед *expr*.
- Для выражения создания объекта DAS для `v` перед arg_1 такое же, как состояние `v` перед *expr*.
- Для каждого аргумента arg_i DAS для `v` перед arg_1 определяется обычными правилами для выражений, игнорируя любые модификаторы `ref` или `out`.

- Для каждого аргумента arg_i, для которого *i* больше 1, DAS для v перед arg_1 такое же, как состояние v после arg_{i-1}.
- Если переменная v передается как аргумент типа `out` (то есть аргумент в форме `out` *v*) для любого из аргументов, то v после *expr* явно присвоена. В других случаях DAS для v после *expr* такое же, как DAS для v после arg_n.
- Для инициализаторов массивов (раздел 7.6.10.4), инициализаторов объектов (раздел 7.6.10.2), инициализаторов коллекций (раздел 7.6.10.3) и инициализаторов анонимных объектов (раздел 7.6.10.6) DAS задается теми элементами, в терминах которых определяются данные конструкции.

5.3.3.23. Выражения простого присваивания

Для выражения *expr* в форме *w* = *expr-rhs*:

- DAS для v до *expr-rhs* такое же, как DAS для v до *expr*.
- Если w является той же переменной, что и v, то v после *expr* является явно присвоенной. В других случаях DAS для v после *expr* такое же, как DAS для v после *expr-rhs*.

5.3.3.24. &&-выражения

Для выражения *expr* в форме *expr-1* && *expr-2*:

- DAS для v перед *expr-1* такое же, как DAS для v перед *expr*.
- Переменная v перед *expr-2* является явно присвоенной, если v после *expr-1* является либо явно присвоенной, либо «явно присвоенной после выражения `true`». В остальных случаях она не является явно присвоенной.
- DAS для v после *expr* определяется следующим:
    - Если v после *expr-1* является явно присвоенной, то v после *expr* является явно присвоенной.
    - Иначе, если v после *expr-2* является явно присвоенной и v после *expr-1* является «явно присвоенной после выражения false», то v после *expr* является явно присвоенной.
    - Иначе, если v после *expr-2* является явно присвоенной или «явно присвоенной после выражения true», то v после *expr* является «явно присвоенной после выражения true».
    - Иначе, если v после *expr-1* является «явно присвоенной после выражения false» и переменная v после *expr-2* является «явно присвоенной после выражения false», то v после *expr* является «явно присвоенной после выражения false».
    - Иначе переменная v после *expr* не является явно присвоенной.

В примере

```
class A
{
    static void F(int x, int y)
```

```
    {
        int i;
        if (x >= 0 && (i = y) >= 0)
{
            // i явно присвоена
        }
        else
        {
            // i не присвоена явно
        }
        // i не присвоена явно
    }
}
```

переменная `i` считается явно присвоенной в одном из вложенных операторов оператора `if`, но не в другом. В операторе `if` метода `F` переменная `i` явно присвоена в первом вложенном операторе, поскольку выполнение выражения `(i = y)` всегда предшествует выполнению этого вложенного оператора. Напротив, переменная `i` не является явно присвоенной во втором вложенном операторе, поскольку `x >= 0` может оказаться `false`, и в результате переменная `i` будет не присвоена.

5.3.3.25. Выражения ||

Для выражения *expr* в форме *expr-1* `||` *expr-2*:

- DAS для `v` перед *expr-1* такое же, как DAS для `v` перед *expr*.
- Переменная `v` перед *expr-2* является явно присвоенной, если `v` после *expr-1* является либо явно присвоенной, либо «явно присвоенной после выражения false». Иначе она не является явно присвоенной.
- DAS для `v` после *expr* определяется следующим:
  - Если `v` после *expr-1* является явно присвоенной, то `v` после *expr* является явно присвоенной.
  - Иначе, если `v` после *expr-2* является явно присвоенной и `v` после *expr-1* является «явно присвоенной после выражения false», то `v` после *expr* является явно присвоенной.
  - Иначе, если DAS для `v` после *expr-1* является «явно присвоенной после выражения false» и состояние `v` после *expr-2* является «явно присвоенной после выражения false», то DAS для `v` после *expr* является «явно присвоенной после выражения false».
  - Иначе переменная `v` после *expr* не является явно присвоенной.

В примере

```
class A
{
    static void F(int x, int y)
    {
        int i;
        if (x >= 0 && (i = y) >= 0)
```

продолжение ⮳

```
{
            // i явно присвоена
        }
        else
        {
            // i не присвоена явно
        }
        // i не присвоена явно
    }
}
```

переменная i считается явно присвоенной в одном из вложенных операторов оператора `if`, но не в другом. В операторе `if` метода `G` переменная `i` явно присвоена в первом вложенном операторе, поскольку выполнение выражения `(i = y)` всегда предшествует выполнению этого вложенного оператора. Напротив, переменная `i` не является явно присвоенной во втором вложенном операторе, поскольку `x >= 0` может оказаться `true`, и в результате переменная `i` будет не присвоена.

5.3.3.26. Выражения !

Для выражения *expr* в форме `!` *expr-operand*:

- DAS для `v` после *expr* определяется следующим:
  - Если `v` после *expr-operand* является явно присвоенной, то `v` после *expr* является явно присвоенной.
  - Если `v` после *expr-operand* не является явно присвоенной, то `v` после *expr* также не является явно присвоенной.
  - Если `v` после *expr-operand* является «явно присвоенной после выражения false», то `v` после *expr* является «явно присвоенной после выражения true».
  - Если `v` после *expr-operand* является «явно присвоенной после выражения true», то `v` после *expr* является явно присвоенной после выражения false».

5.3.3.27. Выражения ??

Для выражения *expr* в форме *expr-1* `??` *expr-2*:

- DAS для `v` перед *expr-1* такое же, как DAS для `v` перед *expr*.
- DAS для `v` перед *expr-2* такое же, как DAS для `v` после *expr-1*.
- DAS для `v` после *expr* определяется следующим:
  - Если *expr-1* является константным выражением (раздел 7.19) со значением `null`, то состояние `v` после *expr* такое же, как состояние `v` после *expr-2*.
- Иначе DAS для `v` после *expr* такое же, как DAS для `v` после *expr-1*.

5.3.3.28. Выражения ?:

Для выражения *expr* в форме *expr-cond ? expr-true* : *expr-false*:

- DAS для `v` перед *expr-cond* такое же, как состояние `v` перед *expr*.

- Переменная `v` перед *expr-true* является явно присвоенной в том и только в том случае, если `v` после *expr-cond* является явно присвоенной или «явно присвоенной после выражения true».
- Переменная `v` перед *expr-false* является явно присвоенной в том и только в том случае, если v после *expr-cond* является явно присвоенной или «явно присвоенной после выражения false».
- DAS для `v` после *expr* определяется следующим:
  - Если *expr-cond* является константным выражением (раздел 7.19) со значением `true`, то состояние `v` после *expr* является таким же, как состояние `v` после *expr-true*.
  - Иначе, если *expr-cond* является константным выражением (раздел 7.19) со значением `false`, то состояние `v` после *expr* является таким же, как состояние `v` после *expr-false*.
  - Иначе, если переменная `v` после *expr-true* является явно присвоенной и после *expr-false* `v` также является явно присвоенной, то переменная `v` после *expr* является явно присвоенной.
  - Иначе переменная `v` после *expr* не является явно присвоенной.

5.3.3.29. Анонимные функции

Для *лямбда-выражения* или *выражения-анонимного-метода expr* с телом *body* (*блок* или *выражение*):

- DAS выходной переменной `v` перед *body* такое же, как состояние `v` перед *expr*. То есть DAS выходных переменных наследуется из контекста анонимной функции.
- DAS выходной переменной `v` после *expr* такое же, как состояние `v` перед *expr*.

  Пример

```
delegate bool Filter(int i);
void F()
{
    int max;
    // Ошибка: max не является явно присвоенной
    Filter f = (int n) => n < max;
    max = 5;
    DoWork(f);
}
```

приводит к ошибке компиляции, поскольку `max` не является явно присвоенной в том месте, где объявлена анонимная функция. Пример

```
delegate void D();
void F() {
    int n;
    D d = () => { n = 1; };
    d();
    // Ошибка: n не является явно присвоенной
    Console.WriteLine(n);
}
```

также приводит к ошибке компиляции, поскольку присваивание значения переменной `n` в анонимной функции не действует на состояние `n` вне анонимной функции.

5.4. Ссылки на переменные

Ссылка-на-переменную является *выражением*, которое классифицируется как переменная. *Ссылка-на-переменную* указывает на ячейку хранения, которая может быть доступна как для того, чтобы извлечь текущее значение, так и для того, чтобы сохранить новое значение.

ссылка-на-переменную:
  выражение

В C и C++ *ссылка-на-переменную* известна как *lvalue*.

5.5. Атомарность ссылок на переменные

Чтение и запись следующих типов данных атомарны (неделимы): `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `uint`, `int`, `float` и ссылочные типы. Добавим, что чтение и запись перечислимых типов с базовым типом из предыдущего списка также атомарны. Чтение и запись других типов, в том числе `long`, `ulong`, `double` и `decimal`, так же как и определенных пользователем типов, не являются гарантированно атомарными. За исключением библиотечных функций, разработанных для этих целей, нет гарантий, что процесс чтения-изменения-записи будет атомарным, так же как и в случае инкремента.

ЭРИК ЛИППЕРТ

Авторы описания делают определенные предположения о своих читателях — например, что читатели, которые беспокоятся относительно атомарного характера, всегда знают, что это такое. Если вы не являетесь одним из них, то дело вот в чем: если один поток запишет значение типа `long` 0x0123456776543210 в переменную, содержащую в это время ноль, то может наступить момент, когда другой поток прочтет из этой переменной 0x0123456700000000. Это присваивание не является атомарным, поскольку оно имеет две явно выраженные фазы: сначала верхние 32 бита, затем нижние 32 бита (или наоборот). Язык C# гарантирует, что такое не случится с 32-битовыми типами данных, но не дает гарантий для более длинных типов данных.

ДЖОН СКИТ

Часто путают атомарный характер и другие сложные аспекты моделей памяти. В частности, даже если присваивание значения атомарно, это не означает, что это обязательно будет видно сразу (или вообще будет видно) другим потокам, до того как будут привлечены такие средства синхронизации, как изменяемые переменные, блокировки или явные барьеры памяти.

Глава 6

Приведение типов

Приведение типов позволяет представить выражение как имеющее определенный тип. Приведение дает возможность представить выражение некоторого типа данных в виде другого типа данных или выражение, не имеющее никакого типа, в виде определенного типа. Приведение типов может быть неявным или явным, и это определяет, требуется ли явное приведение. Например, приведение из типа `int` к типу `long` выполняется неявно, следовательно, любое выражение целого типа можно неявно рассматривать как имеющее тип длинного целого. А вот обратное приведение, из `long` к `int`, всегда должно быть явным, для него требуется выполнить операцию явного приведения типов.

```
int a = 123;
long b = a;      // Неявное приведение типов из int к long
int c = (int)b;  // Явное приведение типов long к int
```

Некоторые приведения типов определяются в языке программирования, а также в программах можно определить собственные операции приведения типов (см. раздел 6.4).

ЭРИК ЛИППЕРТ

Разделение приведения типов на «явное» и «неявное» полезно, когда разработчику требуется знать, может ли данное приведение вызвать ошибку на этапе выполнения программы. Ни одно из неявных приведений типов не может вызвать ошибку, а вот явное приведение типов может.

Другая классификация приведения типов, которая в данной книге не рассматривается подробно, это *сохраняющие представление объекта* и *изменяющие представление объекта*. Например, явное приведение из базового класса `Животное` к классу-наследнику `Жираф` могло бы потерпеть неудачу на этапе выполнения, если исходное выражение не является экземпляром класса `Жираф`. Если приведение типа в данном случае происходит без ошибки, можно быть уверенным, что в результате этой операции мы получим такой же экземпляр класса, каким был исходный. Неявное приведение типа из `int` к `double` всегда будет успешным, и всегда изменит представление объекта. Будет создано совершенно новое значение `double`, у которого будет абсолютно другое представление, чем у исходного целого числа.

Эти различия в приведении типов станут важными, когда мы будем рассматривать ковариантное приведение массивов.

6.1. Неявные приведения типов

Следующие операции приведения являются неявными:

- Тождественное преобразование.
- Неявное приведение арифметических типов.
- Неявное приведение перечислений.
- Неявное приведение обнуляемых типов.
- Приведение константы `null`.
- Неявное приведение ссылочных типов.
- Упаковка.
- Неявное динамическое приведение типов.
- Неявное приведение константных выражений.
- Определенное пользователем неявное приведение.
- Преобразование анонимной функции.
- Преобразование группы методов.

Неявное приведение типов может возникать при многих операциях, в том числе при вызове функционального элемента (раздел 7.5.4), приведении типов в выражениях (раздел 7.7.6), присваивании значений (раздел 7.17).

Предопределенное неявное приведение типов всегда производится успешно и никогда не вызывает исключений. Правильно написанное определенное пользователем приведение типов также должно удовлетворять этому правилу.

Для операций приведения типы данных `object` и `dynamic` считаются эквивалентными. Однако приведение для типа `dynamic` (разделы 6.1.8 и 6.2.6) применяется только в выражениях, имеющих тип данных `dynamic` (раздел 4.7).

6.1.1. Тождественные преобразования

Тождественное преобразование конвертирует любой тип данных в такой же тип. Такое преобразование существует для того, чтобы можно было утверждать, что исходная сущность, которая уже имеет результирующий тип данных, преобразуема к этому же типу.

Поскольку типы данных объект `object` и `dynamic` являются эквивалентными, то и преобразование между типами данных `object` и `dynamic` является тождественным преобразованием. Это же относится и к приведению сконструированных типов, которые будут одинаковыми, если заменить все вхождения `dynamic` на `object`.

ВЛАДИМИР РЕШЕТНИКОВ

Тождественное преобразование симметрично. Если тип `T` тождественно преобразуется к типу `S`, то тип `S` также тождественно преобразуется к типу `T`. Если два типа тождественно преобразуются один в другой, то на этапе выполнения программы они будут представлены

одинаковым образом. Например, типы `List<object[]>` и `List<dynamic[]>` на этапе выполнения программы будут представлены как `List<object[]>`.

БИЛЛ ВАГНЕР

Это важное свойство типа данных `dynamic`. Это означает, что такие типы данных, как `List<object>` и `List<dynamic>`, имеют тождественное преобразование. Вы можете преобразовать коллекцию элементов типа `object` к коллекции элементов типа `dynamic`. Но тождественное преобразование между `dynamic` и любым наследником `object` невозможно. Например, невозможно тождественное преобразование между типами `List<string>` и `List<dynamic>`.

6.1.2. Неявные приведения арифметических типов

Неявное приведение *арифметических типов* данных включает в себя следующие преобразования:

- Из `sbyte` к `short`, `int`, `long`, `float`, `double` и `decimal`.
- Из `byte` к `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`.
- Из `short` к `int`, `long`, `float`, `double` и `decimal`.
- Из `ushort` к `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`.
- Из `int` к `long`, `float`, `double` и `decimal`.
- Из `uint` к `long`, `ulong`, `float`, `double` и `decimal`.
- Из `long` к `float`, `double` и `decimal`.
- Из `ulong` к `float`, `double` и `decimal`.
- Из `char` к `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`.
- Из `float` к `double`.

Приведение из `int`, `uint`, `long` и `ulong` к типу `float` и из `long` или `ulong` к типу `double` могут привести к потере точности исходного значения, но не могут вызвать потерю самого значения. Другие же неявные приведения арифметических типов данных не вызывают потерю какой-либо информации.

Не существует неявных приведений к типу `char`, поэтому любые целые типы автоматически не приводятся к `char`.

6.1.3. Неявные приведения перечислений

Неявное приведение перечислений позволяет преобразовать *целый-десятичный-литерал* 0 к любому из *перечислений*, а также к любому *обнуляемому-типу*, базовым для которого является *перечисление*. В последнем случае приведение будет выполнено к перечислению с последующей упаковкой результата (раздел 4.1.10).

> **ЭРИК ЛИППЕРТ**
>
> Это свойство особенно важно для перечислений, представляющих собой набор флагов. Для совместимости с принципами CLR перечисления, используемые как флаги, желательно помечать атрибутом `[Flags]` и задать значение элемента «None» как ноль. А для перечислений, не удовлетворяющих данным критериям, желательно иметь возможность присваивать ноль без выполнения преобразований.
>
> Для более полной информации об использовании перечислений смотрите главу 4.8 книги «Разработка инфраструктуры проектирования: соглашения, идиомы и шаблоны для многократно используемых библиотек .NET» («Framework Design Guidelines»).
>
> Реализация языка C# от Microsoft позволяет использовать *любую* константу со значением ноль в любом перечислении, а не только *литерал* ноль. Это незначительное отступление от спецификации .Net существует по историческим причинам.

> **ДЖОЗЕФ АЛБАХАРИ**
>
> Значением по умолчанию для перечислений является ноль, соответственно присваивание переменной типа `enum` значения 0 устанавливает ее в значение по умолчанию. Для перечислений, являющихся флагами, значение 0 означает «все флаги сброшены».

6.1.4. Неявные приведения обнуляемых типов

Предопределенное неявное приведение обнуляемых типов выполняется по тем же правилам, что и для соответствующих им необнуляемых типов. Для каждого встроенного неявного тождественного или числового преобразования из некоторого необнуляемого типа `S` к необнуляемому типу `T` существуют соответствующие неявные преобразования:

- Неявное приведение из `S?` к `T?`.
- Неявное приведение из `S` к `T?`.

Неявное приведение для обнуляемого типа, основанное на соответствующем приведении из типа `S` к типу `T`, выполняется следующим образом:

- Если выполняется приведение из `S?` к `T?`:
  - Если исходное значение равно `null` (свойство `HasValue` равно `false`), результатом будет значение `null` типа `T?`.
  - Иначе сначала будет выполнено развертывание `S?` в тип `S`, далее выполнено приведение из типа `S` к типу `T` и затем обертывание (wrapping, раздел 4.1.10) `T` для получения типа `T?`.
- Если выполняется приведение из `S` к `T?`, то сначала будет выполнено приведение из типа `S` к типу `T` и затем обертывание `T` для получения типа `T?`.

6.1.5. Неявные приведения литерала null

Для литерала `null` возможно неявное приведение к любому обнуляемому типу. Такое приведение порождает значение `null` (раздел 4.1.10) результирующего обнуляемого типа.

6.1.6. Неявные приведения ссылочных типов

Неявное приведение ссылочных типов может быть следующим:

- Из любого *ссылочного-типа* к `object` или `dynamic`.
- Из любого класса `S` к любому классу `T`, если `S` является наследником `T`.
- Из любого класса `S` к любому интерфейсу `T`, если `S` реализует интерфейс `T`.
- Из любого интерфейса `S` к любому интерфейсу `T`, если `S` является наследником интерфейса `T`.
- Из любого массива `S`, имеющего элементы типа S_E, к массиву `T`, имеющему элементы типа T_E, если соблюдено все из перечисленного далее:
  - массивы `S` и `T` различаются только типом элементов. Иными словами, `S` и `T` имеют одинаковую размерность;
  - типы S_E и T_E являются ссылочными;
  - возможно неявное приведение из S_E к T_E.

БИЛЛ ВАГНЕР

То, что и S_E и T_E должны быть ссылочными типами, означает, что преобразование массивов с элементами арифметических типов недопустимо.

ВЛАДИМИР РЕШЕТНИКОВ

На самом деле компилятор C# от Microsoft не проверяет, что типы S_E и T_E являются ссылочными, он только проверяет, что операция приведения из S_E к T_E является операцией неявного приведения **ссылочных** типов. Это отличие очень незначительно, но оно позволяет выполнять неявное приведение массивов параметров-типов. Обратите внимание, что в примере ниже параметр-тип `T` формально *не является ссылочным типом данных* (раздел 10.1.5), однако на этапе выполнения программы он всегда представляется как ссылочный тип:

```
T[] Foo<T,S>(S[] x) where S: class, T
{
   return x;
}
```

ЭРИК ЛИППЕРТ

Ковариантные приведения массивов были спорным дополнением к CLI и C#. Такие преобразования нарушают безопасность типов для массивов. Вы можете подумать, что

продолжение ☞

> возможно добавить элемент типа `Turtle` к массиву `Animals`, но возможно выполнить неявное приведение массива `Giraffes` к выражению, тип которого является `Animals`. Если вы попытаетесь добавить элемент типа `Turtle` к такому массиву, на этапе выполнения среда не допустит приведение `Turtle` к `Giraffes`. Таким образом, в некоторых случаях *недопустимо* добавлять элемент типа `Turtle` в массив `Animals`, и вы сможете узнать об этом только на этапе выполнения программы. Компилятор не способен распознать ошибку такого рода.
> Для таких случаев было бы полезным разделять приведение типов на сохраняющие и изменяющие представление исходного значения. Правила CLI для ковариантных приведений требуют, чтобы приведение из исходного типа к требуемому типу сохраняло представление значения на этапе выполнения программы. Например, приведение массива из десяти целых чисел к массиву из десяти чисел с плавающей точкой требует выделения памяти для десяти чисел с плавающей точкой, и значит, оно не может быть быстро выполнено «на месте». И наоборот, приведение массива `Giraffes` к массиву `Animals` сохраняет представление каждого элемента массива на этапе выполнения программы, а значит, также сохраняет представление самого массива.
> Так как приведения ссылочных типов всегда сохраняют представление объекта, язык C# предусматривает явные и неявные ковариантные приведения массивов только если преобразование между типами элементов массивов являются ссылочными. CLR допускает и другие сохраняющие представление приведения, например из массива `int[]` к массиву `uint[]`.

- Из любого *массива* к `System.Array` или интерфейсам, которые он реализует.
- Из любого одномерного массива `S[]` к интерфейсу `System.Collections.Generic.IList<T>` и его базовым интерфейсам, при условии что существует неявное тождественное или ссылочное преобразование из `S` к `T`.
- Из любого *делегата* к типу `System.Delegate` или интерфейсам, которые он реализует.
- Из литерала `null` к любому *ссылочному типу*.
- Из любого *ссылочного-типа* к другому *ссылочному-типу* `T`, если исходный тип имеет неявное тождественное или ссылочное преобразование к ссылочному типу T_0, а T_0 имеет тождественное преобразование к `T`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Например, существует неявное ссылочное приведение из `List<object>` к `ICollection<dynamic>`.

- Из любого *ссылочного-типа* к интерфейсу или делегату `T`, если есть неявное тождественное или ссылочное преобразование к интерфейсу или делегату T_0, и T_0 является типом, приводимым вариантно (раздел 13.1.3.2) к `T`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Например, существует неявное ссылочное приведение из `List<string>` к `IEnumerable<object>`.

- Неявное приведение, включающее параметры-типы, которые заведомо являются ссылочными. Более подробно об этом в разделе 6.1.10.

Неявные приведения ссылочных типов всегда производятся успешно, следовательно, не требуют дополнительной проверки на этапе выполнения.

Приведения ссылочных типов никогда не меняют сам объект. Иными словами, приведение ссылочных типов может изменить тип ссылки, но никогда не изменит тип или значение объекта, на который ссылается.

6.1.7. Преобразования упаковки

Преобразование упаковки позволяет неявно приводить *тип-значение* к ссылочному типу. Упаковка возможна для любого *необнуляемого-типа-значения* в `object` или `dynamic`, в `System.ValueType` и в любой *интерфейс*, реализованный этим *необнуляемым типом-значением*. *Перечисление* может быть приведено к типу `System.Enum`.

Упаковка возможна из *обнуляемого типа* в ссылочный тип тогда и только тогда, когда она возможна из базового *необнуляемого типа-значения* в этот ссылочный тип.

Тип-значение может быть преобразован с помощью упаковки к интерфейсу `I`, если возможна упаковка этого типа к интерфейсу I_0 и для интерфейса I_0 существует тождественное преобразование к интерфейсу `I`.

Тип-значение может быть преобразован с помощью упаковки к интерфейсу `I`, если возможна упаковка этого типа к интерфейсу или делегату I_0 и интерфейс I_0 является приводимым вариантно (раздел 13.1.3.2) к `I`.

Упаковка *необнуляемого типа-значения* состоит в создании экземпляра класса `object` и копировании *значения* типа-значения в этот созданный экземпляр класса.

Структуры могут быть упакованы в тип `System.ValueType`, так как это базовый класс для всех структур (раздел 11.3.2).

Упаковка *обнуляемого типа* происходит следующим образом:

- Если исходное значение равно `null` (свойство `HasValue` равно `false`), результатом будет ссылка `null` нужного типа.
- В противном случае результатом будет ссылка на упакованный `T`, созданный последовательным развертыванием и обертыванием исходного значения.

Более подробно упаковка описана далее в разделе 6.3.1.

6.1.8. Неявные приведения типа dynamic

Возможно неявное приведение из выражения, имеющего тип `dynamic`, к любому типу `T`. Приведение относится к динамическому связыванию (раздел 7.2.2). Это означает, что на этапе выполнения программы будет выполнен поиск неявного приведения из типа времени выполнения этого выражения к типу `T`; если ни одно приведение не найдено, будет выброшено исключение.

ВЛАДИМИР РЕШЕТНИКОВ

Этот факт не означает, что возможно неявное приведение из **типа** `dynamic` к любому типу `T`. Эта разница важна для некоторых вариантов перегрузки методов:

```
class A
{
   static void Foo(string x) { }
   static void Foo(dynamic x) { }
   static void Main()
   {
      Foo(null);
   }
}
```

Перегруженный вариант `Foo(string x)` «лучше», чем `Foo(dynamic x)`, потому что существует неявное приведение из `string` к `dynamic`, но *не* из `dynamic` к `string`.

Стоит отметить, что данное неявное приведение на первый взгляд нарушает правило, приведенное в начале раздела 6.1, что неявное приведение типов не должно выбрасывать исключение. Однако здесь выбрасывает исключение не само приведение типов, а *поиск* возможного неявного приведения. Риск генерации исключения на этапе выполнения программы следует из самой идеи динамического связывания. Если динамическое связывание при приведении типов нежелательно, то выражение можно сначала привести к `object`, а уже затем к нужному типу. Вот пример неявного приведения типа `dynamic`:

```
object o = "object"
dynamic d = "dynamic";
string s1 = o;    // Ошибка на этапе выполнения программы: не существует
                  // приведения типов
string s2 = d;    // Компилируется и успешно выполняется
int i = d;        // Компилируется, но на этапе выполнения программы
                  // возникает ошибка: не существует приведения типов
```

Операции присваивания значений переменным `s2` и `i` используют неявное приведение типа `dynamic`, когда связывание будет выполнено только на этапе выполнения программы. На этапе выполнения будет произведен поиск неявного приведения из типа времени выполнения переменной `d` (`string`) к требуемым типам. Приведение к `string` будет найдено, а к `int` — нет.

6.1.9. Неявные приведения константных выражений

Неявное приведение константных выражений производится для следующих случаев:

- *Константное-выражение* типа `int` (раздел 7.13) может быть приведено к типам `sbyte`, `byte`, `short`, `ushort`, `uint` и `ulong`, если значение *константного-выражения* находится в пределах допустимого диапазона для результирующего типа.

> **ДЖОН СКИТ**
>
> В этом C# более гибок, чем Java. Java предусматривает приведение константных выражений только при их присваивании. Например, следующий код допустим в C#, но эквивалентный Java-код компилироваться не будет, так как число 10 нельзя привести к `byte`:
>
> ```
> void MethodTakingByte(byte b) { ... }
> ...
> // В другом методе
> MethodTakingByte(10);
> ```

- *Константное-выражение* типа `long` может быть приведено к типу `ulong`, если значение *константного-выражения* не отрицательное.

6.1.10. Неявные приведения параметров-типов

Существуют следующие неявные приведения для параметра-типа `T`:

- Из `T` к его эффективному базовому классу `C`, из `T` к любому базовому классу `C`, из `T` к любому интерфейсу, реализуемому классом `C`. Во время выполнения программы, если `T` является типом-значением, приведение выполняется как упаковка, иначе оно выполняется как неявное тождественное или ссылочное преобразование.
- Из `T` к интерфейсу `I` из эффективного набора интерфейсов класса `T`, из `T` к любому базовому по отношению к `I` интерфейсу. Во время выполнения программы, если `T` является типом-значением, приведение выполняется как упаковка, иначе оно выполняется как неявное тождественное или ссылочное преобразование.
- Из `T` к параметру-типу `U`, если `T` зависит от `U` (раздел 10.1.5). Во время выполнения программы, если `U` — тип-значение, то `U` и `T` одного типа, и приведение не выполняется. Иначе, если `T` — тип-значение, приведение выполняется как упаковка. В противном случае оно выполняется как неявное тождественное или ссылочное преобразование.
- Из литерала `null` к типу `T`, если `T` — ссылочный тип.
- Из `T` к ссылочному типу `S`, если существует неявное приведение к ссылочному типу S_o и S_o имеет тождественное преобразование к типу `S`. Во время выполнения программы приведение выполняется таким же образом, как и приведение к типу S_o.
- Из `T` к интерфейсу `I`, если существует неявное приведение к интерфейсу I_o и I_o может быть вариантно приведен к `I` (раздел 13.1.3.2). Во время выполнения программы, если `T` — тип-значение, приведение выполняется как упаковка. Иначе оно выполняется как неявное тождественное или ссылочное преобразование.

Если `T` — ссылочный тип (раздел 10.1.5), то все преобразования, описанные выше, классифицируются как неявные ссылочные преобразования (раздел 6.1.6). Если `T` *не* относится к ссылочным типам, то они классифицируются как преобразования упаковки (раздел 6.1.7).

6.1.11. Определенные пользователем неявные приведения типов

Определенные пользователем неявные преобразования включают в себя необязательное стандартное неявное приведение, следующее за ним определенное пользователем неявное приведение и последующее другое стандартное приведение (необязательно). Более подробно правила выполнения определенного пользователем неявного приведения описаны в разделе 6.4.4.

6.1.12. Приведения анонимной функции и группы методов

Анонимные функции и группы методов не имеют типа сами по себе, но могут быть приведены к типу делегата или дерева выражений. Приведение анонимных функций более детально описано в разделе 6.5, а приведение группы методов в разделе 6.6.

6.2. Явные приведения типов

Следующие операции приведения относятся к явным:

- Все неявные приведения.
- Явные приведения арифметических типов.
- Явные приведения перечислений.
- Явные приведения обнуляемых типов.
- Явные приведения ссылочных типов.
- Явные приведения к интерфейсам.
- Преобразование распаковки.
- Явные приведения типа `dynamic`.
- Определенные пользователем явные приведения типов.

Явные приведения типов возникают в выражениях преобразования типа (раздел 7.7.6).

В число явных приведений типов входят все неявные. Это означает, что каждая операция неявного приведения может быть выполнена как явная.

Явные приведения в отличие от неявных не всегда завершаются успешно, они могут вызывать потерю информации. Приведения между различными типами достаточно разнообразны и заслуживают подробного изложения.

6.2.1. Явные приведения арифметических типов

Явные приведения *арифметических-типов* представляют собой такие приведения из одного *арифметического-типа* в другой, для которых не существует неявных приведений (раздел 6.1.2):

- Из sbyte к byte, ushort, uint, ulong и char.
- Из byte к sbyte и char.
- Из short к sbyte, byte, ushort, uint, ulong и char.
- Из ushort к sbyte, byte, short и char.
- Из int к sbyte, byte, short, ushort, uint, ulong и char.
- Из uint к sbyte, byte, short, ushort, int и char.
- Из long к sbyte, byte, short, ushort, int, uint, ulong и char.
- Из ulong к sbyte, byte, short, ushort, int, uint, long и char.
- Из char к sbyte, byte и short.
- Из float к sbyte, byte, short, ushort, int, uint, long, ulong, char и decimal.
- Из double к sbyte, byte, short, ushort, int, uint, long, ulong, char, float и decimal.
- Из decimal к sbyte, byte, short, ushort, int, uint, long, ulong, char, float и double.

Так как явные приведения включают в себя все явные и неявные операции приведения, следовательно, всегда можно выполнить операцию приведения из любого *арифметического-типа* к любому другому *арифметическому-типу* (раздел 7.7.6).

Явное приведение типов может стать причиной потери информации или вызвать исключение. Явное приведение типов выполняется следующим образом:

- Для приведений из одного целочисленного типа к другому целочисленному типу процесс зависит от контекста контроля переполнения (раздел 7.6.12), в котором происходит приведение:
  - В проверяемом контексте приведение выполняется успешно, если исходное значение находится в допустимых для результирующего типа пределах, и выбрасывается исключение System.OverflowException, если значение вне этих пределов.
  - В непроверяемом контексте приведение типов всегда выполняется успешно и происходит следующим образом:
    - Если исходный тип данных больше, чем результирующий, то исходное значение обрезается путем отбрасывания «лишних» старших битов. Полученный результат рассматривается как значение результирующего типа.
    - Если исходный тип данных меньше, чем результирующий, то он «расширяется» до размера результирующего типа, с учетом того, имело ли исходное значение знак или было беззнаковым. Полученный результат рассматривается как значение результирующего типа.
    - Если размер исходного типа данных такой же, как результирующего типа данных, то исходное значение рассматривается как значение результирующего типа.
- При приведении из типа decimal к любому целочисленному типу исходное значение округляется до целого числа (отбрасыванием дробной части), и это

целое число становится результатом операции. Если получившееся целое число выходит за пределы допустимых значений для результирующего типа, выбрасывается исключение `System.OverflowException`.

- При приведении из `float` или `double` к любому целочисленному типу процесс зависит от контекста контроля переполнения (раздел 7.6.12) для данного приведения типов:
  - В проверяемом контексте приведение происходит следующим образом:
    - Если исходное значение `NaN` или бесконечность, выбрасывается исключение `System.OverflowException`.
    - Иначе исходное значение округляется до целого числа (отбрасыванием дробной части). Если полученное целое число находится в пределах допустимых значений результирующего типа, то это число будет результатом операции приведения типов.
    - Иначе выбрасывается исключение `System.OverflowException`.
  - В непроверяемом контексте приведение типов всегда выполняется успешно и производится следующим образом:
    - Если исходное значение `NaN` или бесконечность, то результатом будет случайное число результирующего типа.
    - Иначе значение округляется до целого числа (отбрасыванием дробной части). Если полученное целое число находится в пределах допустимых значений результирующего типа, то это число будет результатом операции приведения типов.
    - Иначе результатом будет случайное число результирующего типа.
- Для приведения из типа `double` к типу `float` исходное значение типа `double` округляется до ближайшего значения типа `float`. Если значение `double` слишком мало, чтобы быть представленным как `float`, результатом будет или +0, или –0. Если значение типа `double` слишком велико, чтобы быть представленным как `float`, то результатом будет или +∞, или –∞. Если значением `double` является `NaN`, то и результатом будет `NaN`.
- Для приведения из типа `float` или `double` к типу `decimal` исходное значение округляется и приводится к ближайшему числу `decimal` (раздел 4.1.7). Если исходное значение слишком мало, чтобы быть представленным как `decimal`, результатом будет нуль. Если исходное значение является `NaN`, бесконечностью или слишком велико, чтобы быть представленным как `decimal`, выбрасывается исключение `System.OverflowException`.
- Для приведения из типа `decimal` к типам `float` или `double` исходное значение округляется до ближайшего значения типа `float` или `double`. При таком приведении типов может быть потеряна точность значения, но оно никогда не вызывает исключения.

ДЖОЗЕФ АЛЬБАХАРИ

Отбрасывание дробной части при приведении чисел с десятичной точкой к целым числам — рациональный, но не всегда удобный способ. Например, в соответствии

> с этим, число (int) 3,9 будет преобразовано в значение 3, а не 4. C# не предоставляет встроенных механизмов приведения к ближайшему целому числу, эти возможности предусмотрены в дополнительных библиотеках. В Microsoft Framework.NET есть статический класс Convert, обеспечивающий данную возможность округления с помощью своих методов, например ToInt32.

6.2.2. Явные приведения перечислений

Явные приведения перечислений могут быть следующими:

- Из sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double и decimal к любому *перечислению*.
- Из любого *перечисления* к sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double и decimal.
- Из любого *перечисления* к любому другому *перечислению*.

При явном приведении перечислений между собой они обрабатываются как их соответствующие базовые типы данных, то есть как явные или неявные приведения арифметических типов. Например, для некоторого перечисления E с базовым типом int приведение из E к byte выполняется как явное приведение (раздел 6.2.1) из int к byte, а приведение из byte к перечислению E производится как неявное приведение (раздел 6.1.2) из byte к int.

6.2.3. Явные приведения обнуляемых типов

Явное приведение обнуляемых типов дает возможность применять операции явного приведения необнуляемых типов к соответствующим им обнуляемым типам. Для каждого предопределенного явного приведения какого-либо необнуляемого типа S к какому-либо необнуляемому типу T (разделы 6.1.1—6.1.3, 6.2.2) существует соответствующее приведение обнуляемых типов:

- Явное приведение из типа S? к типу T?.
- Явное приведение из типа S к типу T?.
- Явное приведение из типа S? к типу T.

Приведение обнуляемых типов, основанное на соответствующем приведении из типа S к типу T, выполняется следующим образом:

- Для приведения из S? к T?:
  - Если исходное значение равно null (свойство HasValue равно false), результатом будет значение null типа T?.
  - Иначе, при приведении типов исходное значение сначала разворачивается из S? в S, затем выполняется приведение из типа S к типу T, и результат обертывается в тип T?.
- При приведении из типа S к типу T? сначала выполняется приведение из S к T, затем результат обертывается в тип T?.

- При приведении из типа S? к типу T сначала выполняется развертывание S? в S, а затем — приведение из S к типу T.

При попытке развернуть значение null, хранящееся в обнуляемом типе, выбрасывается исключение.

6.2.4. Явные приведения ссылочных типов

Явные приведения ссылочных типов бывают следующими:

- Из object или dynamic к любому ссылочному типу данных.
- Из любого класса S к любому классу T, при условии что S является родительским классом T.
- Из любого класса S к любому интерфейсу T, если класс S не является бесплодным классом и не реализует интерфейс T.
- Из любого интерфейса S к любому классу T, если T не является бесплодным классом или если T реализует интерфейс S.
- Из любого интерфейса S к любому интерфейсу T, если S не является производным от T.
- Из любого массива S с элементами S_E, к любому массиву T с элементами T_E, если выполняются сразу все следующие условия:
    - Массивы S и T отличаются только типами своих элементов, то есть массивы S и T имеют одинаковую размерность.
    - И S_E, и T_E являются ссылочными типами.
    - Существует явное приведение из S_E к T_E.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> На самом деле, компилятор C# от Microsoft не проверяет, что типы S_E и T_E являются *ссылочными*, он проверяет только, что приведение из S к T является явным **ссылочным** приведением. Более подробно это описано в разделе 6.1.6.

- Из типа System.Array или интерфейсов, им реализуемых, к любому массиву.
- Из одномерного массива S[] к System.Collection.Generic.IList<T> или его базовым интерфейсам, если существует явное ссылочное приведение из S к T.
- Из System.Collection.Generic.IList<T> или его базовым интерфейсам к одномерному массиву S[], при условии что существует явное тождественное или ссылочное преобразование из S к T.
- Из System.Delegate и реализуемым им интерфейсам к любому *делегату*.
- Из ссылочного типа к ссылочному типу T, если исходный тип имеет явное ссылочное приведение к типу T_0, а тип T_0 имеет тождественное приведение к T.

- Из ссылочного типа к интерфейсу или делегату T, если исходный тип имеет явное ссылочное приведение к интерфейсу или делегату T_0, и либо T_0 приводим вариантно к типу T, либо наоборот (раздел 13.1.3.2).
- Из D<S_1….S_n> к D<T_1….T_n>, где D<X_1….X_n> является обобщенным типом делегата, а D<S_1….S_n> не совместим с D<T_1….T_n> или не тождественен ему, и для каждого параметра-типа X_i в D верно следующее:
  - Если X_i является инвариантным, то S_i тождественно T_i.
  - Если X_i является ковариантным, то существует явное или неявное тождественное или ссылочное преобразование из S_i к T_i.
  - Если X_i является контравариантным, то S_i и T_i либо тождественны, либо являются ссылочными типами.

> **ЭРИК ЛИППЕРТ**
>
> Например, есть переменная типа Action<S>, где S имеет ссылочный тип и содержит экземпляр класса Action<object>. Можно *неявно* привести Action<object> к типу Action<T>, где T — любой ссылочный тип, поэтому должна быть возможность *явно* привести Action<S> к Action<T> для любых ссылочных типов S и T.

- Явное приведение, включающее параметры-типы ссылочных типов. Более подробно это приведение описано в разделе 6.2.6.

Явные приведения ссылочных типов требуют проверки на корректность на этапе выполнения программы.

Чтобы приведение ссылочных типов на этапе выполнения программы завершилось успешно, исходное значение должно быть null, или фактический тип объекта, на который ссылается исходный операнд, должен быть приводимым к результирующему типу либо с помощью неявного приведения ссылочных типов (раздел 6.1.6), либо с помощью упаковки (раздел 6.1.7). Если явное приведение ссылочных типов завершилось ошибкой, будет выброшено исключение System.InvalidCastException.

Приведение ссылочных типов, явное или неявное, никогда не меняет сам объект, на который ссылаются. Другими словами, приведение ссылочных типов меняет тип ссылки, но никогда не меняет тип или значение объекта, адресуемого этой ссылкой.

6.2.5. Преобразование распаковки

Преобразование распаковки позволяет явно преобразовать ссылочный тип в *тип-значение*. Распаковка существует для типов object, dynamic, System.ValueType для приведения их к любому *необнуляемому-типу-значению*, а также для любых интерфейсов для приведения их к *необнуляемому-типу-значению*, если этот тип реализует данный интерфейс. Кроме того, можно выполнить распаковку для типа System.Enum для приведения его к любому перечислению.

Распаковка возможна из ссылочного типа в *обнуляемый-тип*, если можно осуществить распаковку из этого ссылочного типа в *необнуляемый-тип-значение,* базовый для обнуляемого.

Распаковка из любого интерфейса `I` к типу-значению `S` существует, если есть распаковка из интерфейса I_0, и I_0 имеет тождественное преобразование к `I`.

Интерфейс `I` может быть распакован к типу-значению `S`, если есть такой интерфейс или делегат I_0, из которого можно выполнить распаковку, и I_0, приводимый вариантно к интерфейсу `I`, или наоборот (раздел 13.1.3.2).

Операция распаковки включает в себя проверку, хранит ли объект в себе упакованное значение результирующего типа-значения. Затем значение копируется в результирующий тип. Распаковка ссылки `null` в *обнуляемый-тип* дает значение `null` результирующего типа. Значение типа `System.ValueType` может быть распаковано к структуре, так как этот тип является базовым для всех структур (раздел 11.3.2).

Распаковка также описана в разделе 4.3.2.

6.2.6. Явные приведения типа dynamic

Явное приведение применяется для приведения значения типа `dynamic` к любому типу `T`.

Приведение типа `dynamic` динамически связано, это значит, что на этапе выполнения программы будет выполнен поиск явного приведения из типа времени выполнения этого выражения к типу `T`; если ни одно приведение не найдено, будет выброшено исключение.

Если динамическое связывание при приведении типов нежелательно, выражение можно сначала привести к типу `object`, а уже затем к нужному типу.

Представим, что некий класс `C` описан следующим образом:

```
class C
{
   int i;
   public C(int i) { this.i = i; }
   public static explicit operator C(string s)
   {
      return new C(int.Parse(s));
   }
}
```

Следующий пример показывает явные приведения типа `dynamic`:

```
object o = "1";
dynamic d = "2";
var c1 = (C)o;    // Компилируется, но возникает ошибка явного
                  // приведения ссылочных типов
var c2 = (C)d;    // Компилируется, и определенное пользователем приведение
                  // выполняется успешно
```

Приведение значения `o` к типу `C` на этапе компиляции считается приведением ссылочного типа. Оно вызовет ошибку на этапе выполнения программы, потому что строка «`1`» в переменной `o` на самом деле не имеет тип `C`. Приведение переменной `d`

к типу `C` — это явное приведение типа `dynamic`, и оно будет успешно выполнено на этапе выполнения программы, так как будет найдено определенное пользователем явное приведение типов из `string` к типу `C`.

6.2.7. Явное приведение параметров-типов

Существуют следующие приведения для данного параметра-типа `T`:

- Из эффективного базового для `T` класса `C` к классу `T` и из любого базового для `C` класса к классу `T`. На этапе выполнения программы, если `T` — тип-значение, приведение выполняется как распаковка. Иначе данное приведение выполняется как тождественное или явное ссылочное преобразование.
- Из любого интерфейса к типу `T`. На этапе выполнения программы, если `T` — тип-значение, приведение выполняется как распаковка. Иначе данное приведение типов выполняется как тождественное или явное ссылочное преобразование.
- Из значения типа `T` к *интерфейсу* `I`, если не существует неявного приведения из типа `T` к интерфейсу `I`. На этапе выполнения программы, если `T` — тип-значение, приведение выполняется как упаковка, а затем как явное ссылочное преобразование. Иначе данное приведение выполняется как тождественное или явное ссылочное преобразование.
- Из параметра-типа `U` к типу `T`, если `T` зависит от `U` (раздел 10.1.5). На этапе выполнения программы, если `U` — тип-значение, то `U` и `T` одного типа, и приведение не выполняется. Иначе, если `T` — тип-значение, приведение выполняется как распаковка. В противном случае оно выполняется как явное тождественное или ссылочное преобразование.

Если `T` — ссылочный тип данных, все эти преобразования классифицируются как явные ссылочные преобразования (раздел 6.2.4). Если `T` *не* является ссылочным типом, то эти преобразования классифицируются как преобразования распаковки (раздел 6.2.5).

ДЖОЗЕФ АЛБАХАРИ

Приведение параметров-типов может быть пояснено таким примером:

```
static T Cast<T>(object value) { return (T)value; }
```

Эта операция может выполняться как приведение ссылочных типов или как распаковка, но оно никогда не выполняется как приведение арифметических типов или как определенное пользователем приведение типов.

```
string s = Cast<string>("s"); // успешно, это приведение ссылочных типов
int i = Cast<int>(3); // успешно, это распаковка
long l = Cast<long>(3); // InvalidCastException: попытка выполнить распаковку
                        // вместо приведения арифметических типов
```

Точно так же приведение типов работает для метода `System.Linq.Enumerable.Cast`, который является стандартной операцией запроса в реализации LINQ от Microsoft.

Правила, описанные выше, не допускают прямого явного приведения неограниченного параметра-типа к не интерфейсному типу, что может вызвать удивление. Это необходимо для того, чтобы исключить возможность неверного приведения типов и сделать семантику таких приведений прозрачной. Рассмотрим, например, такое описание класса:

```
class X<T>
{
   public static long F(T t)
   {
      return (long)t; // Ошибка
   }
}
```

Если бы прямое явное приведение `t` к типу `int` было допустимо, можно было бы ожидать, что результатом вызова `X<int>.F(7)` будет значение `7L`. Однако это не так, потому что стандартное приведение арифметических типов применяется, только если типы являются арифметическими на этапе связывания. Данный пример должен быть написан таким образом:

```
class X<T>
{
   public static long F(T t)
   {
      return (long)(object)t; // Успешно, но только если T имеет тип long
   }
}
```

Теперь этот код компилируется, но при попытке выполнить вызов `X<int>.F(7)` возникнет исключение на этапе выполнения программы, потому что упакованное значение `int` не может быть непосредственно приведено к типу `long`.

6.2.8. Определенные пользователем явные приведения типов

Определенные пользователем явные приведения типов состоят из последовательности необязательного стандартного явного приведения типов, определенного пользователем явного или неявного приведения типов и еще одного необязательного стандартного приведения типов. Правила выполнения определенного пользователем явного приведения типов описаны подробнее в разделе 6.4.5.

> **ЭРИК ЛИППЕРТ**
>
> «Сэндвич» из приведений, окружающих определенные пользователем, — это явные приведения, следовательно, они могут завершаться неудачно. Поэтому определенные пользователем приведения имеют не одну, а три потенциальные точки сбоя на этапе выполнения программы.
>
> Но даже в этом случае окружающие приведения не могут быть определенными пользователем. Например, если есть определенное пользователем приведение из `X` к `Y` и определенное пользователем приведение из `Y` к `Z`, попытка привести значение типа `X` к типу `Z` закончится неудачно.

> Операция явного приведения типов может вызывать довольно длинную цепочку преобразований. Например, возьмем структуру `Foo` с определенным пользователем приведением типов из `Foo?` к `decimal`. Приведение значения типа `Foo` к типу `int?` будет произведено по цепочке из `Foo` к `Foo?`, затем из `Foo?` к `decimal`, из `decimal` к `int`, и, наконец, из `int` к `int?`.

6.3. Стандартные приведения типов

Стандартные приведения типов — это такие предопределенные преобразования, которые могут являться частью операции приведения, определенной пользователем.

6.3.1. Стандартные неявные преобразования

Следующие неявные преобразования определены как стандартные неявные преобразования:

- Тождественные преобразования (раздел 6.1.1).
- Неявные приведения арифметических типов (раздел 6.1.2).
- Неявные приведения обнуляемых типов (раздел 6.1.4).
- Неявные приведения ссылочных типов (раздел 6.1.6).
- Преобразование упаковки (раздел 6.1.7).
- Неявные приведения константных выражений (раздел 6.1.8).
- Неявные приведения с использованием типов-параметров (раздел 6.1.10).

В стандартные неявные преобразования не входят неявные приведения, определенные пользователем.

6.3.2. Стандартные явные преобразования

Стандартные явные преобразования включают в себя все стандартные неявные преобразования и все явные преобразования, для которых существует обратное стандартное неявное преобразование. Другими словами, если есть стандартное неявное приведение из типа `A` к типу `B`, значит, есть стандартное явное приведение из типа `A` к типу `B` и из типа `B` к типу `A`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Например, предопределенные явные арифметические приведения из `double` к `decimal` и из `decimal` к `double` не являются стандартными явными приведениями типов.

6.4. Приведения типов, определенные пользователем

C# позволяет расширить предопределенные явные и неявные приведения типов, используя **определенные пользователем приведения типов**. Определенное пользователем приведение типов задается с помощью операции приведения типов (раздел 10.10.3) в описании класса или структуры.

> **БИЛЛ ВАГНЕР**
> Приведение между типами данных подразумевает, что они являются в той или иной мере взаимозаменяемыми. Это, в свою очередь, означает, что некоторые из таких типов не являются необходимыми. Если вы написали много операций приведения типов, проверьте, не созданы ли для одной и той же цели дублирующие типы данных.
> При написании операций приведения ссылочных типов убедитесь, что результатом операции приведения является ссылка на исходный объект.

6.4.1. Допустимые приведения типов, определенные пользователем

В C# разрешено описывать только некоторые определенные пользователем приведения типов. Например, нельзя переопределить существующие явные и неявные приведения типов.

Для исходного типа `S` и результирующего типа `T`, если `S` или `T` являются обнуляемыми типами, пусть S_0 и T_0 определяют их базовые типы, иначе S_0 и T_0 равны `S` и `T` соответственно. В классе или структуре допускается вводить приведение типа из `S` к `T`, только если выполняются все следующие условия:

- S_0 и T_0 — это различные типы.
- S_0 или T_0 является классом или структурой, в котором находится определение операции.
- Ни S_0, ни T_0 не являются *интерфейсами*.
- Не существуют приведения типов из `S` к `T` или из `T` к `S`, за исключением преобразований, определенных пользователем.

Ограничения по использованию определенных пользователем типов рассматриваются далее в разделе 10.10.3.

6.4.2. Повышающие (lifted) приведения типов

Если пользователем определено приведение из необнуляемого типа-значения `S` к необнуляемому типу-значению `T`, то существует **операция повышающего (lifted) приведения** из `S?` к `T?`. Она выполняет развертывание `S?` в `S`, затем определенное

пользователем приведение из S к T, а затем обертывание из T к T?, за исключением того, что значение null типа S? приводится к значению null типа T? напрямую.

Повышающие приведения типов классифицируются как явные или неявные таким же образом, как базовые для них определенные пользователем приведения. Название «определенное пользователем приведение типов» применяется как к повышающему приведению типов, так и к соответствующему ему базовому приведению.

6.4.3. Выполнение определенных пользователем приведений

Определенное пользователем приведение преобразует значение из **исходного типа** данных в **результирующий тип** данных. Выполнение определенного пользователем приведения типов основано на поиске наиболее подходящего определенного пользователем приведения для данного исходного и результирующего типов. Этот процесс включает в себя несколько этапов:

- Поиск списка классов и структур, из которых рассматриваются определенные пользователем приведения типов. В этот список входят: исходный тип и его базовые классы и результирующий тип с его базовыми классами (здесь подразумевается, что только классы и структуры могут содержать описание определенных пользователем приведений типов, а типы, не являющиеся классами, не имеют и базовых классов). При этом, если исходный или результирующий тип является *обнуляемым-типом*, на этом этапе используется соответствующий необнуляемый тип.
- Для этого списка типов устанавливается, какие определенные пользователем и повышающие приведения типов могут быть применены. Для успешного применения операции приведения типов необходима возможность выполнения стандартного приведения (раздел 6.3) из исходного типа к типу операнда в операции, а также возможность выполнения стандартного приведения из типа результата операции к результирующему типу.
- Из списка возможных определенных пользователем приведений типов выбирается наиболее подходящая операция. В общем случае выбирается то определенное пользователем приведение, в котором тип операнда «наиболее близок» к исходному типу, а тип результата «наиболее близок» к результирующему. При этом определенному пользователем приведению отдается предпочтение относительно повышающего приведения. Полностью правила выбора наиболее подходящего определенного пользователем приведения типов даны в следующих разделах.

Когда наиболее подходящее приведение типов найдено, то реальное вычисление определенного пользователем приведения выполняется в три этапа:

- Если необходимо, выполняется стандартное приведение из исходного типа к типу операнда определенного пользователем или повышающего приведения.

- Далее выполняется само определенное пользователем приведение типов.
- И наконец, если это необходимо, выполняется стандартное приведение из типа, являющегося результатом определенного пользователем приведения, к результирующему типу.

Выполнение определенного пользователем приведения типов никогда не включает в себя более одного определенного пользователем приведения. Иными словами, приведение из типа `S` к типу `T` никогда не будет вычисляться как определенное пользователем приведение из `S` к `X` и последующее определенное пользователем приведение из `X` к `T`.

Точные определения вычисления определенных пользователем явных и неявных приведений типов даны в следующих разделах. В них используются следующие термины:

- Если существует неявное стандартное приведение (раздел 6.3.1) из типа `A` к типу `B`, и ни `A`, ни `B` не являются *интерфейсами*, то говорят, что `A` **охватывается** типом `B`, а `B` **охватывает** `A`.
- **Наиболее охватывающий тип** в списке типов тот, который охватывает все другие типы в этом списке. Если нет ни одного типа, способного охватить все другие в списке, то говорят, что список не имеет наиболее охватывающего типа. То есть наиболее охватывающий тип в списке — это самый «большой» тип, к которому могут быть неявно приведены все остальные типы в списке.
- **Наиболее охваченный тип** в списке типов тот, который можно охватить любым другим типом из этого списка. Если нет ни одного такого типа, то говорят, что список не имеет наиболее охваченного типа. То есть наиболее охваченный тип можно назвать самым «маленьким» типом в списке, он может быть неявно приведен к любому другому типу в списке.

> **БИЛЛ ВАГНЕР**
>
> Чем больше операций приведения типов вы описываете, тем больше вероятность внести в преобразования неоднозначность. Это может сделать ваш класс более сложным для использования.

6.4.4. Определенные пользователем неявные приведения типов

Определенное пользователем неявное приведение из типа `S` к типу `T` выполняется следующим образом:

- Определить типы S_0 и T_0. Если `S` и `T` — обнуляемые типы, то S_0 и T_0 — соответствующие им необнуляемые типы, в противном случае S_0 и T_0 равны соответственно `S` и `T`.
- Найти список типов `D`, определенные пользователем приведения типов которых рассматриваются. Этот список состоит из S_0 (если S_0 — это класс или

структура), базовых классов S_0 (если S_0 — это класс) и T_0 (если T_0 — это класс или структура).

- Найти список применимых определенных пользователем и повышающих неявных приведений типов U. Такой список состоит из определенных пользователем или повышающих неявных операций приведения типов, объявленных в классах или структурах из списка D, которые преобразуют типы, охватывающие S, к типам, охваченным T. Если список U пуст, преобразование не определено, и выдается ошибка компиляции.
- Найти наиболее подходящий исходный тип S_x для операций из списка U:
  - Если какая-либо операция из списка U выполняет приведение из типа S, то за S_x принимается тип S.
  - Иначе S_x — это наиболее охваченный тип в объединенном наборе исходных типов операций в списке U. Если нельзя найти ровно один наиболее охваченный тип, преобразование неоднозначно, и выдается ошибка компиляции.
- Найти наиболее подходящий результирующий тип T_x для операций из списка U:
  - Если какая-либо из операций в списке U выполняет приведение к T, то за T_x принимается тип T.
  - Иначе, T_x — это наиболее охватывающий тип в объединенном наборе результирующих типов операций в списке U. Если нельзя найти ровно один наиболее охватывающий тип, преобразование неоднозначно, и выдается ошибка компиляции.
- Найти наиболее подходящую операцию приведения:
  - Если список U содержит ровно одну определенную пользователем операцию приведения типов из S_x к T_x, то она и является наиболее подходящей.
  - Иначе, если список U содержит ровно одну операцию повышающего приведения типов из S_x к T_x, то она и является наиболее подходящей.
  - Иначе преобразование неоднозначно, и выдается ошибка компиляции.
- И наконец, применяется преобразование:
  - Если S не является S_x, выполняется стандартное неявное приведение из S к S_x.
  - Выполняется наиболее подходящая операция преобразование из S_x к T_x.
  - Если T_x не является T, выполняется стандартное неявное приведение из T_x к T.

6.4.5. Определенные пользователем явные приведения типов

Определенное пользователем явное приведение из типа S к типу T выполняется следующим образом:

- Определить типы S_0 и T_0. Если S и T — обнуляемые типы, то S_0 и T_0 – соответствующие им необнуляемые типы, в противном случае S_0 и T_0 равны соответственно S и T.
- Найти список типов D, определенные пользователем приведения типов которых рассматриваются. Этот список состоит из S_0 (если S_0 — это класс или структура), базовых классов S_0 (если S_0 — это класс), T_0 (если T_0 — это класс или структура) и базовых классов T_0 (если T_0 — это класс).
- Найти список применимых определенных пользователем и повышающих неявных приведений типов U. Такой список состоит из определенных пользователем или повышающих неявных и явных операций приведения типов, объявленных в классах или структурах из списка D, которые преобразуют типы, охватывающие или охваченные S, к типам, охватывающим или охваченным T. Если список U пуст, преобразование не определено, и выдается ошибка компиляции.
- Найти наиболее подходящий исходный тип S_x для операций из списка U:
  - Если какая-либо операция из списка U выполняет приведение из типа S, то за S_x принимается тип S.
  - Иначе, если какая-либо операция из списка U выполняет приведение из типов, которые охватывают S, то S_x — это наиболее охваченный тип среди входных типов для операций приведения типов в списке U. Если нельзя найти ровно один наиболее охваченный тип, преобразование неоднозначно, и выдается ошибка компиляции.
  - Иначе S_x — это наиболее охватывающий тип в объединенном наборе исходных типов операций в списке U. Если нельзя найти ровно один наиболее охватывающий тип, преобразование неоднозначно, и выдается ошибка компиляции.
- Найти наиболее подходящий результирующий тип T_x для операций в списке U:
  - Если какая-либо из операций в списке U выполняет приведение к T, то за T_x принимается тип T.
  - Иначе, если какая-либо операция из списка U выполняет приведение из типов, которые охватываются T_x, — это наиболее охватывающий тип в объединенном наборе результирующих типов операций в списке U. Если нельзя найти ровно один наиболее охватывающий тип, преобразование неоднозначно, и выдается ошибка компиляции.
  - Иначе T_x — это наиболее охваченный тип в объединенном наборе результирующих типов операций в списке U. Если нельзя найти ровно один наиболее охваченный тип, преобразование неоднозначно, и выдается ошибка компиляции.
- Найти наиболее подходящую операцию приведения:
  - Если список U содержит ровно одну определенную пользователем операцию приведения типов из S_x к T_x, то она и является наиболее подходящей.
  - Иначе, если список U содержит ровно одну операцию повышающего приведения типов из S_x к T_x, то она и является наиболее подходящей.
  - Иначе преобразование неоднозначно, и выдается ошибка компиляции.

- И наконец, применяется преобразование:
  - Если `S` не является S_x, выполняется стандартное явное приведение из `S` к S_x.
  - Выполняется наиболее подходящая определенная пользователем операция преобразование из S_x к T_x.
  - Если T_x не является `T`, выполняется стандартное явное приведение из T_x к `T`.

6.5. Приведения анонимных функций

Выражение-анонимного-метода и *лямбда-выражение* классифицируется как анонимная функция (раздел 7.15). Это выражение не имеет типа, но может быть неявно приведено к совместимому типу делегату или к типу дерева выражений. Делегат `D` совместим с анонимной функцией `F` при условиях:

- Если `F` содержит *сигнатуру-анонимной-функции*, то `D` и `F` имеют одинаковое количество параметров.
- Если `F` не содержит *сигнатуру-анонимной-функции*, `D` может не иметь параметров или иметь несколько параметров любого типа, если ни один из них не помечен модификатором `out`.
- Если `F` имеет список параметров с явным указанием их типа, каждый параметр делегата `D` имеет тот же тип и модификаторы, что и соответствующий параметр в `F`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Из этого правила есть одно исключение: если делегат `D` имеет параметр с модификатором `params`, то для соответствующего ему параметра в функции `F` не должно быть никакого модификатора.

- Если `F` имеет список неявно типизированных параметров, делегат `D` не имеет параметров с модификаторами `ref` или `out`.
- Если делегат `D` имеет тип возврата `void`, а тело функции `F` — это выражение, и каждый параметр `F` того же типа, что и соответствующий ему параметр `D`, тело `F` является допустимым выражением (раздел 7), которое может быть представлено в виде *оператора-выражения* (раздел 8.6).
- Если делегат `D` имеет тип возврата `void`, а тело функции `F` — это блок операторов, и каждый параметр `F` того же типа, что и соответствующий ему параметр делегата `D`, тело функции `F` является допустимым блоком операторов (раздел 8.2), оператор `return` в котором не содержит выражения.
- Если делегат `D` имеет тип возврата не `void`, а тело функции F — это выражение, и каждый параметр `F` того же типа, что и соответствующий ему параметр `D`, тело

функции F является допустимым выражением (раздел 7), которое может быть неявно приведено к типу возвращаемого значения D.

- Если D имеет тип возврата не `void`, а тело функции F — это блок операторов, и каждый параметр F того же типа, что и соответствующий ему параметр делегата D, то тело F является допустимым блоком операторов (раздел 8.2) с недоступной конечной точкой, в которой каждый оператор `return` определяет выражение, неявно приводимое к типу возвращаемого значения D.

Дерево выражений `Expression<D>` совместимо с анонимной функцией F, если делегат D совместим с F.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Только анонимные функции в виде *лямбда-выражений* могут иметь неявные приведения к типам деревьев выражений (*выражения-анонимного-метода* к этим типам привести нельзя). Но даже существующие приведения из лямбда-выражений могут вызвать ошибку при компиляции программы.

Некоторые анонимные функции не могут быть приведены к деревьям выражений, даже если соответствующее приведение типов *существует*, оно может вызвать ошибку на этапе компиляции программы. Это происходит, если анонимное выражение содержит одну или несколько из следующих конструкций:

- Простую или сложную операцию присваивания.
- Динамически связанное выражение.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Другие конструкции, не поддерживаемые в текущей реализации, в том числе вложенные *выражения-анонимного-метода*, *лямбда-выражения* с телом операторов, *лямбда-выражения* с параметрами `ref` и `out`, *доступ-к-базовому-классу*, инициализаторы многомерных массивов, именованные и необязательные параметры, неявные ссылки, операции с указателями.
> Если необходимо привести выражение типа `dynamic` к другому типу в дереве выражений, можно использовать либо операцию `as`, либо предварительно привести выражение к типу `object`.

В данном примере используется обобщенный делегат `F<A, R>`, который представляет функцию, принимающую аргумент типа A и возвращающую значение типа R:

```
delegate R Func<A,R>(A arg);
```

В присваиваниях

```
Func<int,int> f1 = x => x + 1;    // Успешно
Func<int,double> f2 = x => x + 1; // Успешно
Func<double,int> f3 = x => x + 1; // Ошибка
```

типы параметров и возвращаемого значения каждой анонимной функции определяются типом переменной, которой присваивается значение анонимной функции.

Первое присваивание успешно приводит анонимную функцию к делегату `Func<int, double>`, поскольку `x` имеет тип `int`, а `x+1` — это допустимое выражение, которое неявно приводимо к типу `int`.

Также и второе присваивание успешно приводит анонимную функцию к делегату `Func<int, double>`, поскольку результат сложения `x+1` типа `int` неявно приводим к типу `double`.

Однако третье присваивание приводит к ошибке на этапе компиляции программы, поскольку `x` имеет тип `double`, и `x+1` тоже будет иметь тип `double`, который невозможно неявно привести к типу `int`.

Анонимные функции могут влиять на разрешение перегрузки и могут участвовать в выведении типов. Более подробно об этом сказано в разделе 7.5.

6.5.1. Выполнение приведений анонимных функций к типам делегатов

Приведение анонимной функции к делегату дает экземпляр делегата, ссылающегося на анонимную функцию, и набор (возможно, пустой) захваченных внешних переменных, которые активны во время вычисления. Когда вызывается делегат, выполняется тело анонимной функции. Код, находящийся в теле функции, выполняется с использованием захваченных внешних переменных, на которые ссылается делегат.

Список вызовов делегата, полученный из анонимной функции, имеет единственный элемент. Точный целевой объект и целевой метод данного делегата не определены. В частности, неизвестно, равен ли объект в делегате значению `null`, значению `this` объемлющего функционального элемента или какому-либо другому объекту.

Приведения семантически идентичных анонимных функций с одинаковым набором (возможно, пустым) экземпляров захваченных внешних переменных к одинаковым типам делегатов допускают (но не требуют) возвращать одинаковые экземпляры делегата. Термин «семантически идентично» используется здесь для обозначения того, что выполнение анонимных функций будет всегда давать одинаковые результаты для одинаковых аргументов. Это правило позволяет оптимизировать код, аналогичный приведенному далее:

```
delegate double Function(double x);
class Test
{
   static double[] Apply(double[] a, Function f)
   {
      double[] result = new double[a.Length];
      for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
      return result;
   }
   static void F(double[] a, double[] b)
   {
      a = Apply(a, (double x) => Math.Sin(x));
      b = Apply(b, (double y) => Math.Sin(y));
      ...
   }
}
```

Так как два делегата анонимных функций имеют тот же (пустой) набор внешних переменных и анонимные функции семантически идентичны, компилятор позволяет делегатам ссылаться на один и тот же метод. Действительно, компилятор позволяет возвращать в точности тот же экземпляр делегата из обоих выражений анонимной функции.

ДЖОН СКИТ

Эта *возможная* оптимизация иногда опасна, если происходит добавление и удаление делегатов, что имеет важное значение для событий.
Рассмотрим такой вариант кода:

```
button.Click += (sender, args) => Console.WriteLine("Clicked");
button.Click -= (sender, args) => Console.WriteLine("Clicked");
```

Представим простейшую реализацию обработки событий. Что будет результатом такого кода? Либо событие `Click` будет иметь те же обработчики, что и до того, либо оно будет иметь один дополнительный обработчик. Будьте осторожны с «оптимизацией», значительно изменяющей поведение программы.

ВЛАДИМИР РЕШЕТНИКОВ

Компилятор C# не кэширует экземпляры делегатов с обобщенными методами.

6.5.2. Выполнение приведений анонимных функций к деревьям выражений

Приведение анонимной функции к дереву выражений дает в результате дерево выражений (раздел 4.6). Говоря более точно, выполнение приведения анонимной функции приводит к созданию структуры объекта, представляющего структуру самой анонимной функции. Точная структура дерева выражений, как и точный процесс его создания, определяется реализацией.

ВЛАДИМИР РЕШЕТНИКОВ

Так же как и делегаты, дерево выражений может иметь набор захваченных внешних переменных.

6.5.3. Пример реализации

В этом разделе описана возможная реализация приведений анонимных функций в терминах других конструкций языка C#. Описываемая реализация основана на принципах, используемых в компиляторе C# от Microsoft. Но это не означает, что такая реализация рекомендуется или является единственно верной.

Ниже рассмотрено несколько примеров кода, которые содержат анонимные функции с различными характеристиками. Для каждого примера приведен соот-

ветствующий перевод в код, использующий только другие конструкции языка C#. В примерах D представляет собой следующий делегат:

```
public delegate void D();
```

Простейшая форма анонимной функции не захватывает внешних переменных:

```
class Test
{
   static void F()
   {
      D d = () => { Console.WriteLine("test"); };
   }
}
```

Этот код может быть преобразован в инстанцирование делегата, который ссылается на сгенерированный компилятором статический метод, в который помещена анонимная функция:

```
class Test
{
   static void F()
   {
      D d = new D(__Method1);
   }
   static void __Method1()
   {
      Console.WriteLine("test");
   }
}
```

В следующем примере анонимная функция ссылается на элементы экземпляра `this`:

```
class Test
{
   int x;
   void F()
   {
   D d = () => { Console.WriteLine(x); };
   }
}
```

Этот код может быть преобразован в сгенерированный компилятором метод экземпляра, содержащий код анонимной функции:

```
class Test
{
   int x;
   void F()
   {
      D d = new D(__Method1);
   }
   void __Method1()
   {
      Console.WriteLine(x);
   }
}
```

В этом примере анонимная функция захватывает локальную переменную:

```
class Test
{
   void F()
   {
      int y = 123;
      D d = () => { Console.WriteLine(y); };
   }
}
```

Время жизни локальной переменной должно быть расширено, по крайней мере, до времени жизни делегата анонимной функции. Это может быть достигнуто «подъемом» локальной переменной в поле созданного компилятором класса. Инстанцирование локальной переменной (раздел 7.15.5.2) в таком случае выполняется при создании экземпляра порожденного компилятором класса, а доступ к локальной переменной осуществляется как к полю порожденного компилятором класса. Кроме того, анонимная функция становится методом порожденного компилятором класса:

```
class Test
{
   void F()
   {
      __Locals1 __locals1 = new __Locals1();
      __locals1.y = 123;
      D d = new D(__locals1.__Method1);
   }
   class __Locals1
   {
      public int y;
      public void __Method1()
      {
         Console.WriteLine(y);
      }
   }
}
```

И, наконец, анонимная функция захватывает `this` и две локальные переменные с различным временем жизни:

```
class Test
{
   int x;

   void F() {
      int y = 123;
      for (int i = 0; i < 10; i++) {
         int z = i * 2;
         D d = () => { Console.WriteLine(x + y + z); };
      }
   }
}
```

Здесь для каждого блока операторов, в котором объявлены локальные переменные, создается порожденный компилятором класс, так чтобы локальные переменные в разных блоках могли иметь независимые времена жизни. Экземпляр

класса `__Locals2`, созданного компилятором для внутреннего блока операторов, содержит локальную переменную `z` и поле, ссылающееся на экземпляр класса `__Locals1`. Экземпляр класса `__Locals1`, созданного компилятором для внешнего блока операторов, содержит локальную переменную `y` и поле, ссылающееся на `this` объемлющего функционального элемента. С использованием таких структур данных ко всем захваченным внешним переменным можно получить доступ через экземпляр класса `__Locals2`, а код анонимной функции может быть реализован как метод класса.

БИЛЛ ВАГНЕР

Если `F()` возвратила экземпляр `D`, время жизни всех переменных, которые захватила эта функция, будет увеличено:

```
class Test
{
   void F()
   {
      __Locals1 __locals1 = new __Locals1();
      __locals1.__this = this;
      __locals1.y = 123;
      for (int i = 0; i < 10; i++)
      {
         __Locals2 __locals2 = new __Locals2();
         __locals2.__locals1 = __locals1;
         __locals2.z = i * 2;
         D d = new D(__locals2.__Method1);
      }
   }
   class __Locals1
   {
      public Test __this;
      public int y;
   }
   class __Locals2
   {
      public __Locals1 __locals1;
      public int z;
      public void __Method1()
      {
         Console.WriteLine(__locals1.__this.x + __locals1.y + z);
      }
   }
}
```

Тот же подход, который был применен для локальных переменных, можно использовать при приведении анонимных функций к деревьям выражений. Ссылки на порожденные компилятором объекты могут быть сохранены в дереве выражений, а доступ к локальным переменным может быть представлен как доступ к полям этих объектов. Преимущества такого подхода в том, что он позволяет делегатам и деревьям выражений совместно использовать «поднятые» локальные переменные.

ЭРИК ЛИППЕРТ

У такого подхода есть и оборотная сторона, которая, к сожалению, присутствует в большинстве реализаций языков с близкой семантикой и заключается в следующем: две разные анонимные функции в одном и том же методе, использующие две различные локальные переменные, вызывают увеличение времени жизни обеих переменных до времени жизни наиболее «долгоживущего» делегата.

Предположим, у вас есть две локальные переменные `expensive` и `cheap` и два делегата, каждый из которых использует одну из переменных:

```
D longlived = ()=>cheap;
D shortlived = ()=>expensive;
```

Время жизни `expensive` будет не короче, чем время жизни объекта, на который ссылается `longlived`, даже при том, что `expensive` на самом деле в `longlived` не используется. В более изощренной реализации переменные были бы разделены по разным порожденным компилятором классам, и этой проблемы удалось бы избежать.

6.6. Приведения групп методов

Существует неявное приведение (раздел 6.1) из группы методов (раздел 7.1) к совместимому делегату. Рассмотрим делегат `D` и выражение `E`, которое классифицируется как группа методов. Неявное приведение из `E` к `D` существует, если `E` состоит, по меньшей мере, из одного метода, который применим в его нормальной форме (раздел 7.5.3.1) к списку параметров, сконструированному с использованием параметров-типов и модификаторов делегата `D`, как описано далее.

Как производится приведение из группы методов `E` к делегату `D` на этапе компиляции, описано ниже. Заметим, что существование неявного приведения из `E` к `D` не гарантирует, что приведение на этапе компиляции пройдет без ошибок.

ВЛАДИМИР РЕШЕТНИКОВ

В частности, возможно, что в результате разрешения перегрузки будет выбран наиболее подходящий функциональный элемент, содержащий в сигнатуре тип делегата, а неявное приведение из группы методов к этому типу делегата, переданной как соответствующий аргумент, приведет к ошибке компиляции. Такая ситуация может возникать, например, из-за того, что разрешение перегрузки не сможет найти единственный лучший метод в этой группе. Даже если наиболее подходящий метод найден, ограничения типов его параметров могут быть нарушены, или метод может быть не совместим с делегатом, или это метод экземпляра, на который ссылаются из статического контекста.

- Одиночный метод `M` выбирается в соответствии с вызовом метода (раздел 7.6.5.1) в форме `E(A)` со следующими модификациями:
    - Список параметров `A` — это список выражений, каждое из которых классифицируется как переменная, имеющая такой же тип и модификаторы доступа (`ref` или `out`), как соответствующий параметр в *списке-формальных-параметров* делегата `D`.

  - Из возможных методов учитываются только те, которые применимы в своей нормальной форме (раздел 7.5.3.1), а не те, которые применимы только в расширенной форме.
- Если алгоритм раздела 7.6.5.1 заканчивается ошибочно, то возникает ошибка компиляции. Иначе этот алгоритм выберет лучший метод `M`, имеющий то же количество параметров, что и делегат `D`, и приведение считается существующим.
- Выбранный метод `M` должен быть совместим (раздел 15.2) с делегатом `D`, иначе возникает ошибка компиляции.
- Если выбранный метод `M` — это метод экземпляра, то выражение экземпляра, связанное с `E`, будет определять целевой объект делегата.
- Если выбранный метод `M` — это метод расширения, к которому обращаются через выражение экземпляра, то это выражение определяет целевой объект делегата.
- Результатом приведения является значение типа `D`, а именно вновь созданный делегат, который ссылается на выбранный метод и целевой объект.

Отметим, что этот процесс может привести к созданию делегата с методом расширения, если алгоритм раздела 7.6.5.1 не сможет найти метод экземпляра, но сможет создать вызов `E(A)` как вызов метода расширения (раздел 7.6.5.2). Созданный таким образом делегат захватывает метод расширения и его первый аргумент.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Если первый параметр метода расширения, преобразованный к типу делегата, является типом-значением, или параметр-тип не является ссылочным типом (раздел 10.1.5), возникает ошибка компиляции.
>
> Если метод экземпляра, объявленный в `System.Nullable<T>`, приводится к делегату, возникает ошибка компиляции.

Следующий пример демонстрирует приведение группы методов.

```
delegate string D1(object o);
delegate object D2(string s);
delegate object D3();
delegate string D4(object o, params object[] a);
delegate string D5(int i);
class Test
{
   static string F(object o) {...}
   static void G()
   {
      D1 d1 = F; // Правильно
      D2 d2 = F; // Правильно
      D3 d3 = F; // Ошибка: не применим
      D4 d4 = F; // Ошибка: не применим в нормальной форме
      D5 d5 = F; // Ошибка: применим, но не совместим
   }
}
```

Присваивание переменной `d1` неявно приводит группу методов `F` к значению типа `D1`.

Присваивание переменной `d2` показывает, как можно создать делегат для метода, который имеет менее унаследованные (контравариантные) типы параметра и более унаследованный (ковариантный) тип возвращаемого значения.

Присваивание переменной `d3` показывает, как соответствующее приведение типов не существует, если метод не применим.

Присваивание переменной `d4` показывает, как метод должен быть применим в его нормальной форме.

Присваивание переменной `d5` показывает, что типы параметров и возвращаемого значения делегата и метода могут различаться только для ссылочных типов.

Так же, как и для всех других явных и неявных приведений, для явного приведения группы методов может использоваться операция приведения. В данном примере

```
object obj = new EventHandler(myDialog.OkClick);
```

код можно написать по-другому.

```
object obj = (EventHandler)myDialog.OkClick;
```

Группы методов могут влиять на разрешение перегрузки и участвовать в выведении типа. Эти возможности более детально описаны в разделе 7.5.

Во время выполнения программы приведение группы методов производится следующим образом:

- Если метод, выбранный на этапе компиляции, является методом экземпляра или если это метод расширения, доступ к которому осуществляется как к методу экземпляра, то целевой объект делегата определяется по выражению экземпляра, связанному с `E`:
  - Выражение экземпляра вычисляется, если при этом возникает исключение, а дальнейшие шаги не выполняются.
  - Если выражение экземпляра имеет *ссылочный-тип*, то значение, полученное из выражения экземпляра, становится целевым объектом делегата. Если выбранный метод является методом экземпляра и целевой объект имеет значение `null`, выбрасывается исключение `System.NullReferenceException` и дальнейшие шаги не выполняются.
  - Если выражение экземпляра имеет *тип-значение*, то производится упаковка (раздел 4.3.1) к типу `object`, и этот объект становится целевым для делегата.
- Иначе выбранный метод является частью вызова статического метода, и целевой объект делегата имеет значение `null`.
- Создается новый экземпляр делегата `D`. Если для создания нового экземпляра недостаточно памяти, то выбрасывается исключение `System.OutOfMemoryException` и дальнейшие шаги не выполняются.

> **ДЖОН СКИТ**
>
> В спецификации утверждается, что будет создан новый экземпляр, и это не позволяет использовать оптимизацию. Компилятор C# от Microsoft способен кэшировать делегаты, созданные с помощью анонимных функций, если они не захватывают никаких переменных, включая `this`. Такой же способ кэширования мог бы использоваться для делегатов, созданных посредством приведения группы методов, выбирающего статический метод, но спецификация это запрещает.

- Новый экземпляр делегата инициируется ссылкой на метод, который был определен на этапе компиляции, и ссылкой на целевой объект, определенный выше.

Глава 7
Выражения

Выражение представляет собой последовательность операций и операндов. В этой главе рассказывается о синтаксисе и значении выражений, порядке вычисления операндов и операций.

7.1. Классификация выражений

Выражения классифицируются следующим образом:

- Значение. Каждое значение имеет связанный с ним тип.
- Переменная. Каждая переменная имеет связанный с ней тип, а именно объявленный тип переменной.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Типы локальных переменных и параметров лямбда-выражений могут во многих случаях определяться компилятором и указывать их типы в объявлениях необходимо не всегда.

- Пространство имен. Выражения этого вида могут появляться только в левой части *доступа-к-элементу* (раздел 7.6.4). В любом другом контексте выражение, классифицированное как пространство имен, приводит к ошибке компиляции.
- Тип. Выражения этого вида могут появляться только в левой части *доступа-к-элементу* (раздел 7.6.4) или как операнд операции `as` (раздел 7.10.11), операции `is` (раздел 7.10.10) или операции `typeof` (раздел 7.6.11). В любом другом контексте выражение, классифицированное как тип, приводит к ошибке компиляции.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Если выражение классифицировано как тип и появляется в левой части *доступа-к-элементу*, оно никогда не может обозначать тип массива или указателя. Если выражение обозначает параметр типа, это всегда в дальнейшем вызывает ошибку компиляции.

- Группа методов, которая представляет собой ряд перегруженных методов, являющихся результатом поиска элемента (раздел 7.4). Группа методов мо-

жет иметь связанное с ней выражение экземпляра и связанный с ней список аргументов типа. Когда вызывается метод экземпляра, результат вычисления выражения экземпляра становится экземпляром, представленным с помощью операции `this` (раздел 7.6.7). Группа методов допускается в *выражении-вызова* (*invocation-expression*) (раздел 7.6.5), в выражении *создания-делегата* (раздел 7.6.10.5) и в левой части оператора `is` и может быть неявно приведена к совместимому типу делегата (раздел 6.6). В любом другом контексте выражение, классифицированное как группа методов, приводит к ошибке компиляции.

ЭРИК ЛИППЕРТ

То, что группа методов допустима в левой части оператора `is`, может немного сбивать с толку. Результат вычисления `is` всегда будет `false`, даже если группа методов приводима к типу правой части.

ВЛАДИМИР РЕШЕТНИКОВ

Группу методов в левой части оператора `is` нельзя использовать в деревьях выражений.

- Литерал `null`. Выражение этого вида можно неявно привести к ссылочному или к обнуляемому типу.
- Анонимная функция. Выражение этого вида можно неявно привести к совместимому типу делегата или к типу дерева выражений.

ЭРИК ЛИППЕРТ

Группы методов, анонимные функции и литерал `null` — все это выражения, не имеющие типа. Такие необычные выражения могут использоваться только тогда, когда тип можно определить из контекста.

- Доступ к свойству. Каждый доступ к свойству имеет связанный с ним тип — а именно, тип свойства. Кроме того, доступ к свойству может иметь связанное с ним выражение экземпляра. Когда вызван код доступа (блок `get` или `set`) к свойству экземпляра, результат вычисления выражения экземпляра становится экземпляром, представленным с помощью `this` (раздел 7.6.7).
- Доступ к событию. Каждый доступ к событию имеет связанный с ним тип, а именно тип события. Кроме того, доступ к событию может иметь связанное с ним выражение экземпляра. Доступ к событию может появиться как левая часть операций `+=` и `-=` (раздел 7.167.3). В любом другом контексте выражение, классифицированное как доступ к событию, приводит к появлению ошибки компиляции.
- Доступ к индексатору. Каждый доступ к индексатору имеет связанный с ним тип, а именно тип элемента индексатора. Кроме того, доступ к индексатору

может иметь связанное с ним выражение экземпляра и связанный с ним список аргументов. Когда вызван код доступа к индексатору (блок `get` или `set`), результат вычисления выражения экземпляра становится экземпляром, представленным с помощью `this` (раздел 7.6.7), и результат вычисления списка аргументов становится списком параметров вызова.

- Пустое выражение. Такое выражение появляется, когда выражение является вызовом метода, имеющего тип возвращаемого значения `void`. Выражение, классифицированное как пустое, допустимо только в контексте *выражения-оператора* (раздел 8.6).

Окончательный результат выражения никогда не является пространством имен, типом, группой методов или доступом к событию. Как уже говорилось, эти категории выражений являются промежуточными конструкциями, которые разрешены только в определенном контексте.

Доступ к свойству или индексатору всегда переклассифицируется в значение при выполнении вызова *кодов доступа get* (*get-accessor*) или *set* (*set-accessor*). Конкретный код доступа определяется контекстом доступа к свойству или к индексатору: если доступ является целевым объектом присваивания, то вызывается *set-accessor* для присваивания нового значения (раздел 7.17.1). Иначе вызывается *get-accessor* для получения текущего значения (раздел 7.1.1).

7.1.1. Значение выражений

В большинстве конструкций, включающих в себя выражения, в конце концов требуется получить **значение** выражения. В таких случаях, если выражение фактически обозначает пространство имен, тип, группу методов или пустое выражение, выдается ошибка компиляции. Однако если выражение обозначает доступ к свойству, доступ к индексатору или переменную, неявно подставляется значение этого свойства, индексатора или переменной:

- Значение переменной есть просто текущее значение, находящееся в ячейке хранения, идентифицируемой этой переменной. Переменные должны считаться явно присвоенными (раздел 5.3) до того, как их значения можно получить; в противном случае выдается ошибка компиляции.
- Значение выражения доступа к свойству получается при вызове кода доступа свойства *get*. Если свойство не имеет этого кода доступа, выдается ошибка компиляции. В противном случае выполняется вызов функционального элемента (раздел 7.5.4), и результат вызова становится значением выражения доступа к свойству.
- Значение выражения доступа к индексатору получается при вызове кода доступа *get* индексатора. Если индексатор не имеет этого кода доступа, выдается ошибка компиляции. В противном случае выполняется вызов функционального элемента (раздел 7.5.4) со списком аргументов, связанных с выражением доступа к индексатору, и результат вызова становится значением выражения доступа к индексатору.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Если свойство или индексатор имеют код доступа *get*, но этот код доступа имеет модификатор доступа и не является достижимым в текущем контексте, выдается ошибка компиляции.

7.2. Статическое и динамическое связывание

Процесс определения значения операции, основывающийся на типе или значении составных частей выражения (аргументов, операндов, получателей), часто называют связыванием. Например, значение вызванного метода определяется в соответствии с типом получателя и аргументов. Значение операции определяется в соответствии с типом ее операндов.

В C# значение операции обычно определяется на этапе компиляции на основании типа времени компиляции выражений, составляющих операцию. Если выражение содержит ошибку, компилятор обнаруживает эту ошибку и сообщает о ней. Такой подход известен как **статическое связывание**.

Однако если выражение является *динамическим* (то есть имеет тип `dynamic`), это означает, что соответствующий процесс связывания должен основываться на типе, определенном во время выполнения программы. Связывание такой операции всегда отложено до того времени, когда операция будет выполняться во время выполнения программы. Такой процесс называется **динамическим связыванием**.

Когда операция является динамически связанной, компилятор практически не выполняет ее проверку. Вместо этого, если связывание во время выполнения программы не приводит к положительному результату, ошибки обрабатываются как исключения.

Следующие операции в C# подлежат связыванию:

- Доступ к элементу: `e.M`.
- Вызов метода: $\texttt{e.M(e}_1\texttt{,...,e}_n\texttt{)}$.
- Вызов делегата: $\texttt{e(e}_1\texttt{,...,e}_n\texttt{)}$.
- Доступ к элементу: $\texttt{e[e}_1\texttt{,...,e}_n\texttt{]}$.
- Создание объекта: $\texttt{new C(e}_1\texttt{,...,e}_n\texttt{)}$.
- Перегруженные унарные операции: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`.
- Перегруженные бинарные операции: `+`, `-`, `*`, `/`, `%`, `&`, `&&`, `|`, `||`, `??`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Операции присваивания значения: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`.
- Явные и неявные преобразования.

Когда динамические выражения отсутствуют, C# по умолчанию настроен на статическое связывание, которое означает, что будут использоваться типы под-

выражений, определенные во время компиляции. Однако если одно из таких выражений в операциях, представленных выше, является динамическим, операция будет динамически связанной.

7.2.1. Время связывания

Статическое связывание происходит во время компиляции, а динамическое связывание происходит во время выполнения программы. В следующих разделах термин **время связывания** относится либо ко времени компиляции, либо ко времени работы программы, в зависимости от того, какой тип связывания имеет место.

Следующий пример иллюстрирует понятия статического и динамического связывания и времени связывания:

```
object o = 5;
dynamic d = 5;

Console.WriteLine(5); // Статическое связывание с  Console.WriteLine(int)
Console.WriteLine(o); // Статическое связывание с Console.WriteLine(object)
Console.WriteLine(d); // Динамическое связывание с Console.WriteLine(int)
```

Первые два вызова статически связаны: перегрузка `Console.WriteLine` выбирается на основе определенного во время компиляции типа аргумента. Таким образом, временем связывания здесь является *время компиляции*.

Третий вызов является динамически связанным: `Console.WriteLine` выбирается на основе определенного во время выполнения программы типа аргумента. Так происходит потому, что аргумент является динамическим выражением — его тип времени компиляции — `dynamic`. Таким образом, временем связывания для третьего вызова является время выполнения программы.

БИЛЛ ВАГНЕР

Многие путают динамическое связывание с выведением типов. Компилятор определяет тип во время компиляции. Например:

```
var i = 5;             // i является int (Компилятор выполняет выведение типов)
Console.WriteLine(i);  // Статическое связывание с  Console.WriteLine(int)
```

Компилятор выводит, что `i` — целое. Все связывание с переменной `i` является статическим.

7.2.2. Динамическое связывание

Цель динамического связывания — позволить программам на C# взаимодействовать с динамическими объектами, то есть с объектами, которые не подчиняются обычным правилам системы типов C#. Динамическими объектами могут быть объекты из других языков программирования с другой системой типов, или это могут быть объекты, которые программно устроены таким образом, чтобы реализовывать собственную семантику связывания для различных операций.

ЭРИК ЛИППЕРТ

Примерами таких объектов, в первую очередь послуживших причиной для реализации динамического связывания, являются: (1) объекты динамических языков, таких как IronPython, IronRuby, JScript и т. д.; (2) объекты объектных моделей «expando», одной из главных особенностей которых является возможность добавления новых свойств во время выполнения программы, например объектная модель документа Internet Explorer (Internet Explorer Document Object Model) и другие объектные модели, использующие разметку (markup-based object models), и (3) унаследованные объекты COM, например в объектной модели Microsoft Office. В данном случае присутствовало стремление облегчить профессиональным разработчикам C# взаимодействие с этими системами; у нас *не было* намерения вообще сделать C# динамическим языком программирования, подобным JScript.

Механизм, с помощью которого динамические объекты реализуют собственную семантику, определяется реализацией. Некоторый интерфейс — опять же, определенный реализацией, — выполняется динамическими объектами, чтобы сигнализировать среде выполнения C#, что они имеют свою особую семантику. Таким образом, когда операции над динамическими объектами динамически связаны, применяется их собственная семантика связывания, которая в данном случае имеет преимущество перед семантикой C#, описанной в этой книге.

ЭРИК ЛИППЕРТ

Механизм реализации этой особенности в Microsoft на самом деле выбран такой же, какой используется в IronPython и других DLR языках программирования для того, чтобы сделать эффективным их динамический анализ и диспетчеризацию. Динамические операции компилируются в вызов методов DLR-объектов, которые затем используют специальные версии компоновщика выражений C# для построения деревьев выражений, представляющих динамические операции. Эти деревья выражений затем компилируются, кэшируются и вновь используются в следующий раз, когда выполняется вызов.

Хотя целью динамического связывания является осуществление взаимодействия с динамическими объектами, C# позволяет производить динамическое связывание со всеми объектами, независимо от того, являются ли они динамическими или нет. Такой подход позволяет сгладить интеграцию динамических объектов, так как результаты операций над ними могут и не являться динамическими объектами, но все же иметь тип, не известный программисту во время компиляции. Также динамическое связывание может помочь исключить подверженный ошибкам код, основанный на рефлексии, даже в том случае, когда он не содержит динамических объектов.

ЭРИК ЛИППЕРТ

C# и ранее поддерживал форму динамической диспетчеризации методов: виртуальные методы с технической точки зрения являются формой динамической диспетчеризации, поскольку для точного определения, какой метод вызывать, используется тип времени выполнения получателя. Хотя мы не имели в виду сделать C# динамическим языком (как,

продолжение ⇗

я уже говорил раньше), существует по меньшей мере одна проблема программирования, когда динамическая диспетчеризация оказывается полезной: проблема «множественной виртуальной диспетчеризации». Эта проблема возникает, когда вы хотите выбрать, какой метод вызвать, основываясь на типе времени выполнения множества аргументов, а не только на типе получателя. (Нетрудно выполнить «двойную виртуальную» диспетчеризацию для паттерна «посетитель», но выполнять во время выполнения программы диспетчеризацию для методов со многими аргументами-типами) неудобно.

В следующих разделах для каждой конструкции языка точно описано, в каких случаях используется динамическое связывание, какой вид проверки во время компиляции (если таковой есть) применяется, что является результатом компиляции и как классифицируются выражения.

7.2.3. Типы составных частей выражений

Когда операция статически связана, типы составных частей выражения (то есть типы получателя, аргумента, индекса и операнда) всегда рассматриваются как типы времени компиляции.

Когда операция динамически связана, типы составляющих выражения определяются различными способами в зависимости от типов времени компиляции составляющих выражения:

- Подвыражение, имеющее тип времени компиляции `dynamic`, считается имеющим тип фактического значения, который будет определен при вычислении выражения во время выполнения программы.

БИЛЛ ВАГНЕР

Правило, что выражения типа времени компиляции `dynamic` используют фактический тип времени выполнения, приводит к удивительному поведению, например к следующей ошибке компиляции:

```
public class Base
{
   public Base(dynamic parameter)
   {
   }
}

public class Derived : Base
{
   public Derived(dynamic parameter) : base(parameter)
   {
   }
}
```

Динамическая диспетчеризация не допускается во время конструирования, так что необходимо использовать статическую диспетчеризацию для вызова базового конструктора.

- Подвыражение, компилированный тип которого является параметром-типом, считается имеющим тип, с которым параметр-тип будет связан во время выполнения программы.
- Иначе составляющая выражения считается имеющей тип времени компиляции.

7.3. Операции

Выражения составляются из операндов и операций. Операции выражения показывают, какие действия производятся с операндами. Примерами операций являются `+`, `-`, `*`,`/` и `new`. Примерами операндов являются константы, поля, локальные переменные и выражения.

Есть три вида операций:

- Унарные операции. Унарные операции используют один операнд и либо префиксную запись (например, `–x`), либо постфиксную запись (например, `x++`).
- Бинарные операции. Все бинарные операции используют два операнда и инфиксную запись (например, `x + y`).
- Тернарные операции. Существует только одна тернарная операция `?:`. Она использует три операнда и инфиксную запись (`c? x: y`).

Порядок действий в выражении определяется приоритетом и ассоциативностью операций (раздел 7.3.1).

Операнды в выражении вычисляются слева направо. Например, в выражении `F(i) + G(i++) * H(i)` вызываемый метод `F` использует старое значение `i`, далее, метод `G` также вызывается со старым значением `i`, и, наконец, метод `H` использует новое значение `i`. Это никак не связано с приоритетом операций.

Некоторые операции могут быть перегружены. Перегрузка операций позволяет задать определенные пользователем реализации операций для операций, в которых один или оба операнда имеют определенный пользователем тип класса или структуры (раздел 7.3.2).

7.3.1. Приоритет и ассоциативность операций

Когда выражение содержит множество операций, приоритет операций управляет порядком, в котором вычисляются отдельные операции. Например, выражение `x + y * z` вычисляется как `x + (y * z)`, потому что операция `*` имеет более высокий приоритет, чем бинарная операция `+`. Приоритет операции устанавливается определением связанного с ним грамматического правила. Например, *аддитивное-выражение* состоит из последовательности *мультипликативных-выражений*, разделенных знаками операций `+` или `-`; таким образом, операциям `+` и `–` присваивается меньший приоритет, чем операциям `*`, `/` и `%`.

Следующая таблица дает краткое описание всех операций в порядке убывания приоритета.

Раздел	Категория	Операция
7.6	Первичные	`x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate`
7.7	Унарные	`+ - ! ~ ++x --x (T)x`
7.8	Мультипликативные	`* / %`
7.8	Аддитивные	`+ -`
7.9	Сдвиг	`<< >>`
7.10	Отношения и проверки типа	`< > <= >= is as`
7.10	Равенство	`== !=`
7.11	Логическое AND	`&`
7.11	Логическое XOR	`^`
7.11	Логическое OR	`|`
7.11	Условное AND	`&&`
7.12	Условное OR	`||`
7.12	Нулевое объединение	`??`
7.14	Условная	`?:`
7.17, 7.15	Присваивание и лямбда-выражения	`= *= /= %= += -= <<= >>= &= ^= |= =>`

Когда операнд находится между двумя операциями одного приоритета, порядок, в котором выполняются операции, определяется ассоциативностью:

- За исключением операций присваивания, все бинарные операции являются *левоассоциативными*, что означает, что операции выполняются слева направо. Например, `x + y + z` вычисляется как `(x + y) + z`.
- Операции присваивания являются *правоассоциативными*, что означает, что операции выполняются справа налево. Например, `x = y = z` вычисляется как `x = (y = z)`.

Приоритетом и ассоциативностью можно управлять при помощи круглых скобок. Например, в выражении `x + y * z` сначала `y` умножается на `z`, а затем результат прибавляется к `x`, но в выражении `(x + y) * z` сначала складываются `x` и `y`, а затем результат умножается на `z`.

КРИС СЕЛЛЗ

Я не являюсь большим поклонником программ, которые полагаются на приоритет операций по умолчанию. Когда порядок выполнения сомнителен, используйте при записи выражений круглые скобки. Круглые скобки не влияют на компилированный вывод (за исключением случаев, когда вам нужно найти ошибку), но делают код гораздо легче для понимания человека, который его читает.

ДЖЕСС ЛИБЕРТИ

Я не вижу ничего страшного в том, чтобы пойти еще дальше и *всегда* записывать выражения в скобках. *Вы* можете не иметь ни малейшего сомнения в том, чего хотите, но бедный программист, который пытается уразуметь, для чего ваш код предназначен, не

> должен гадать, какой приоритет придаст ему смысл. Если число скобок становится удручающим, лучше разбить операцию на несколько, используя временные переменные.

> **ЭРИК ЛИППЕРТ**
>
> Отношения между приоритетом, ассоциативностью, круглыми скобками и порядком выполнения могут сбить с толку. *Операнды* всегда вычисляются строго слева направо. Способ, которым проучаются результаты вычислений, определяется приоритетом, ассоциативностью и круглыми скобками. Подчеркиваю, что это *не тот случай*, когда `y` и `z` вычисляются раньше, чем `x`. Да, умножение выполняется раньше сложения, но вычисление `x` происходит *перед* умножением.

7.3.2. Перегрузка операций

Все унарные и бинарные операции имеют предопределенные реализации, которые автоматически доступны в любом выражении. В добавление к предопределенным реализациям могут быть введены реализации, определенные пользователем, путем включения объявлений операций в классы и структуры (раздел 10.10). Реализации, определенные пользователем, всегда имеют преимущество перед предопределенными реализациями операции: только когда не существует подходящих реализаций, определенных пользователем, рассматриваются преопределенные реализации операции, как это описано в разделах 7.3.3 и 7.3.4.

Перегружаемые унарные операции:

```
+    -    !    ~    ++    --    true    false
```

Хотя `true` и `false` в выражениях не используются явно (и, как следствие, не включены в таблицу приоритетов в разделе 7.3.1), они рассматриваются как операции, поскольку привлекаются в некоторых контекстах выражений: в булевских выражениях (раздел 7.20), в выражениях, включающих в себя условную операцию (раздел 7.14), и в условных логических операциях (раздел 7.12).

Перегружаемые бинарные операции:

```
+  –  *  /  %  &  |  ^  <<  >>  ==  !=  >  <  >  =  <  =
```

Только приведенные выше операции могут быть перегружены. В частности, невозможно перегружать доступ к элементу, вызов метода или операции `=`, `&&`, `||`, `??`, `?:`, `=>`, `checked`, `unchecked`, `new`, `typeof`, `default`, `as` и `is`.

Когда бинарная операция перегружена, соответствующая операция присваивания также неявно перегружается. Например, перегруженная операция `*` означает, что операция `*=` также перегружена. Это описано далее в разделе 7.17.2. Отметим, что сама по себе операция присваивания (`=`) не может быть перегружена. Присваивание всегда выполняется в виде простой побитовой копии значения в переменную.

Операции приведения вида `(T)x` перегружаются с помощью преобразований, определенных пользователем (раздел 6.4).

Доступ к элементу вида `a[x]` не может рассматриваться как перегружаемая операция. Вместо этого индексирование, определенное пользователем, поддерживается с помощью индексаторов (раздел 10.9).

В выражениях операции используются с помощью знаков операций (операторная запись), а в объявлениях — с помощью функциональной записи. Следующая таблица иллюстрирует соотношение между двумя способами записи для унарных и бинарных операций. В первой строке *op* обозначает любую перегружаемую унарную префиксную операцию. Во второй строке *op* обозначает унарные постфиксные операции `++` и `--`. В третьей строке *op* обозначает любую перегружаемую бинарную операцию.

Операторная запись	**Функциональная запись**
op x	**operator** *op*(x)
x *op*	**operator** *op*(x)
x *op* y	**operator** *op*(x, y)

Для объявлений операций, определенных пользователем, всегда требуется, чтобы по меньшей мере один из параметров имел тип класса или структуры, который (которая) содержит объявление операции. Таким образом, определенная пользователем операция не может иметь такую же сигнатуру, как предопределенная операция.

Объявления операций, определенных пользователем, не могут менять синтаксис, приоритет и ассоциативность операции. Например, операция `/` всегда является бинарной, всегда имеет уровень приоритета, определенный в разделе 7.3.1, и всегда является левоассоциативной.

Хотя операции, определенные пользователем, могут выполнять любые вычисления, какие можно пожелать, те реализации, которые дают иные результаты, чем можно ожидать интуитивно, сильно удручают. Например, при выполнении операции `operator ==` должны сравниваться два операнда — равны ли они, и возвращаться соответствующий результат типа `bool`.

БИЛЛ ВАГНЕР

Помимо следования этому правилу следует ограничить использование операций, определенных пользователем, и применять их только в тех случаях, когда операция понятна подавляющему большинству разработчиков, которые будут иметь дело с этим кодом.

ЭРИК ЛИППЕРТ

Все перегружаемые операции являются математическими или логическими по своей природе. Если вы строите библиотеку математических объектов, таких как матрицы, векторы и т. д., то, конечно, используйте операции, определенные пользователем. Но, пожалуйста, избегайте соблазна создать «изящную» операцию типа «Клиент плюс Заказ равняется Счет-фактура».

Описания отдельных операций в разделах 7.6–7.12 подробно описывают предопределенные реализации операций и некоторые дополнительные правила, приме-

нимые к каждой операции. В этих описаниях используются термины: **разрешение перегрузки унарной операции**, **разрешение перегрузки бинарной операции** и **числовое расширение**, определения которых вы найдете в следующих разделах.

7.3.3. Разрешение перегрузки унарных операций

Операция вида *op* `x` или `x` *op*, где *op* есть знак перегруженной унарной операции, а `x` есть выражение типа `X`, обрабатывается следующим образом:

- Набор операций, определенных пользователем для типа `X` для операции `operator` *op*(`x`), определяется применением правил раздела 7.3.5.
- Если набор определенных пользователем операций непустой, он становится набором возможных операций для данной операции. В противном случае набором возможных операций для данной операции становятся предопределенные реализации унарной операции `operator` *op*, в том числе ее расширенные формы. Предопределенные реализации данной операции определены в описании операции (раздел 7.6 и 7.7).
- Правила разрешения перегрузки раздела 7.5.3 применяются к набору возможных операций для выбора наиболее подходящей операции в соответствии со списком аргументов `list (x)`, и эта операция становится результатом процесса перегрузки. Если в результате разрешения перегрузки невозможно выбрать единственную наиболее подходящую операцию, выдается ошибка времени связывания.

7.3.4. Разрешение перегрузки бинарной операции

Операция вида `x` *op* `y`, где *op* есть знак перегружаемой бинарной операции, а `x` есть выражение типа `X` и `y` есть выражение типа `Y`, обрабатываются следующим образом:

- Определяется набор операций, определенных пользователем для типов `X` и `Y` для операции `operator` *op*(`x, y`). Такой набор представляет собой объединение возможных операций для типа `X` и возможных операций для типа `Y`, каждая из которых определяется применением правил раздела 7.3.5. Если `X` и `Y` одного типа или они являются потомками общего родительского типа, то возможные операции появляются в объединенном наборе один раз.
- Если набор определенных пользователем операций непустой, он становится набором возможных операций для данной операции. В противном случае набором возможных операций для данной операции становятся предопределенные реализации унарной операции `operator` *op*, в том числе ее расширенные формы. Предопределенные реализации данной операции определены в описании операции (разделы 7.8–7.12).
- Правила разрешения перегрузки раздела 7.5.3 применимы к набору возможных операций для выбора наиболее подходящей операции в соответствии со списком

аргументов `list (x, y)`, и эта операция становится результатом процесса перегрузки. Если в результате разрешения перегрузки невозможно выбрать единственную наиболее подходящую операцию, выдается ошибка времени связывания.

7.3.5. Набор определенных пользователем операций

Для данного типа `T` и операции `operator` *op*(`A`), где *op* есть знак перегружаемой операции и `A` есть список аргументов, набор определенных пользователем операций для типа `T` для операции `operator` *op*(`A`), определяется следующим образом:

- Определяется тип T_0. Если `T` есть обнуляемый тип, T_0 есть его базовый тип; в противном случае `T` эквивалентен T_0.
- Для всех объявлений операции `operator` *op* в T_0 и всех расширенных форм таких операций, если как минимум одна операция является подходящей (раздел 7.5.3.1) в соответствии со списком аргументов `A`, то набор возможных операций состоит из всех подходящих операций в T_0.
- Иначе, если T_0 является классом `object`, набор возможных операций является пустым.
- Иначе, набор возможных операций типа T_0 есть набор возможных операций прямого базового класса T_0 или эффективного базового класса T_0, если T_0 является параметром-типом.

7.3.6. Числовое расширение

Числовое расширение состоит в автоматическом выполнении некоторых неявных преобразований операндов предопределенных арифметических унарных и бинарных операций. Числовое расширение является не особым механизмом, а скорее эффектом применения разрешения перегрузки предопределенных операций. Числовое расширение не влияет на вычисление операций, определенных пользователями, хотя такие операции могут быть реализованы подобным образом.

В качестве примера числового расширения рассмотрим предопределенные реализации бинарной операции *:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

Когда правила разрешения перегрузки применяются к этому набору операций, выбирается первая из операций, для которой существует неявное преобразование из типов операндов. Например, для операции `b * s`, где `b` является `byte`, а `s` — `short`, разрешение перегрузки выберет в качестве наиболее походящей операцию

operator *(int, int). Таким образом, b и s будут преобразованы к int и тип результата будет int. Точно так же для операции i * d, где i есть int и d являются double, разрешение перегрузки выберет в качестве наиболее подходящей операции operator *(double, double).

7.3.6.1. Унарное числовое расширение

Унарное числовое расширение выполняется для операндов предопределенных унарных операций +, – и ~. Унарное числовое расширение состоит просто в приведении операндов типа sbyte, byte, short, ushort и char к типу int. Кроме того, для унарной операции – унарное числовое расширение приводит операнды типа uint к типу long.

7.3.6.2. Бинарное числовое расширение

Бинарное числовое расширение выполняется для операндов предопределенных операций +, –, *, /, %, &, |, ^, ==, !=, >, <, >= и <=. Бинарное числовое расширение неявно приводит оба операнда к общему типу, который в случае не связанных операций становится также типом результата операции. Бинарное расширение состоит в применении следующих правил, в том порядке, в котором они здесь приведены:

- Если один из операндов имеет тип decimal, другой операнд приводится к типу decimal, или, если другой операнд имеет тип float или double, выдается ошибка времени связывания.
- Иначе, если один из операндов имеет тип double, другой операнд приводится к типу double.
- Иначе, если один из операндов имеет тип float, другой операнд приводится к типу float.
- Иначе, если один из операндов имеет тип ulong, другой операнд приводится к типу ulong, или, если другой операнд имеет тип sbyte, short, int или long, выдается ошибка времени связывания.
- Иначе, если один из операндов имеет тип long, другой операнд приводится к типу long.
- Иначе, если один из операндов имеет тип uint и другой операнд имеет тип sbyte, short, или int, оба операнда приводятся к типу long.
- Иначе, если один из операндов имеет тип uint, другой операнд преобразуется к типу uint.
- Иначе оба операнда приводятся к типу int.

Отметим, что первое правило запрещает любые операции, смешивающие тип decimal с типами double и float. Это правило следует из того факта, что не существует неявных преобразований между типом decimal и типами double и float.

Также отметим, что один из операндов не может быть типа ulong, когда другой операнд является операндом целого типа со знаком. Причина этого состоит в том, что не существует целого типа, который может представлять и полный ряд значений ulong, и значения целых типов со знаком.

В обоих описанных случаях может использоваться выражение приведения типов для явного преобразования одного из операндов к типу, который совместим с другим операндом. В примере

```
decimal AddPercent(decimal x, double percent)
   {
return x * (1.0 + percent / 100.0);
}
```

выдается ошибка времени связывания, поскольку `decimal` нельзя умножать на `double`. Эта ошибка устраняется с помощью явного преобразования второго операнда к типу `decimal`, например, следующим образом:

```
decimal AddPercent(decimal x, double percent)
{
   return x * (decimal)(1.0 + percent / 100.0);
}
```

ДЖОЗЕФ АЛЬБАХАРИ

Предопределенные операции, описанные в этом разделе, всегда активируют 8-битовый и 16-битовый целые типы, а именно `short`, `ushort`, `sbyte` и `byte`. Обычная ловушка здесь состоит в присваивании результата вычисления над этими типами снова 8-битовому или 16-битовому целому типу:

```
byte a = 1, b = 2;
byte c = a + b;           // Ошибка компиляции
```

В этом случае переменные `a` и `b` расширяются до типа `int`, который для успешной компиляции требует явного приведения к типу `byte`.

7.3.7. Расширенные операции

Расширенные (lifted) операции дают возможность применять предопределенные и определенные пользователем операции, применяемые к необнуляемым значимым типам, также к обнуляемым формам этих типов. Расширенные операции сконструированы из предопределенных и определенных пользователем операций, которые соответствуют требованиям, описанным далее:

- Для унарных операций

  `+    ++    -    --    !    ~`

  расширенная форма операции существуют, если типы операнда и результата —необнуляемые значимые типы. Расширенная операция создается путем добавления одного модификатора ? к типам операнда и результата. Результатом операции является нулевое значение, если операнд есть ноль. В противном случае расширенная операция развертывает операнд, применяет базовую операцию и обертывает результат.

- Для бинарных операций

  `+    -    *    /    %    &    |    ^    <<    >>`

расширенная форма операции существует, если типы операндов и результата — необнуляемые значимые типы. Расширенная операция создается путем добавления одного модификатора `?` к типу каждого операнда и к типу результата. Расширенная операция дает в результате нулевое значение, если один или оба операнда имеют значение ноль (за исключением операций `&` и `|`, а также операций типа `bool?`, как описано в разделе 7.11.3). В противном случае расширенная операция развертывает операнд, применяет базовую операцию и обертывает результат.

- Для операций равенства

  `==   !=`

  расширенная форма операции существует, если оба типа операндов являются ненулевыми значимыми типами и если тип результата есть `bool`. Расширенная форма создается путем добавления одного модификатора `?` к каждому типу операнда. Расширенная операция считает два нулевых значения равными, а нулевое значение — не равным любому ненулевому значению. Если оба операнда ненулевые, расширенная операция развертывает операнды и применяет базовую операцию, дающую результат типа `bool`.

- Для операций сравнения

  `<   >   <=   >=`

  расширенная форма операции существует, если оба типа операндов являются ненулевыми значимыми типами и тип результата есть `bool`. Расширенная форма создается путем добавления одного модификатора `?` к каждому типу операнда. Расширенная операция дает в результате значение `false`, если один или оба операнда имеют значение ноль. В противном случае расширенная операция развертывает операнды и применяет базовую операцию, дающую результат типа `bool`.

БИЛЛ ВАГНЕР

Значение по умолчанию любого обнуляемого типа не больше и не меньше, чем значение для необнуляемого типа.

7.4. Поиск элемента

Поиск элемента (member lookup) есть процесс, в котором определяется значение имени в контексте типа. Поиск элемента может выполняться как часть вычисления *простого-имени* (раздел 7.6.2) или *доступа-к-элементу* (раздел 7.6.4) в выражении. Если вычисление простого имени или доступ к элементу осуществляются как *первичное-выражение* или *выражение-вызова* (раздел 7.6.5.1), элемент называется *вызванным* (*invoked*).

Если элемент является методом или событием или если он является константой, полем или свойством типа делегата (глава 15) или типа `dynamic` (раздел 4.7), то он называется *вызываемым* (*invocable*).

При поиске элемента рассматривается не только имя элемента, но также число параметров-типов и возможность доступа к элементу. Для целей поиска элемента обобщенные методы и вложенные обобщенные типы имеют число параметров-типов, указанное в соответствующих объявлениях, а все другие элементы не имеют параметров-типов.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Результат поиска элемента никогда не содержит операций, индексаторов, явно реализованных интерфейсов, статических конструкторов, конструкторов экземпляра, деструкторов (финализаторов) и элементов, создаваемых компилятором. Хотя параметр-тип входит в область объявлений объявляемого типа, он не является элементом этого типа и, таким образом, не может быть результатом поиска элемента.
>
> Поиск элемента возвращает как статические элементы, так и элементы экземпляров, независимо от формы доступа.

Поиск элемента с именем `N` с `K` параметрами-типами в типе `T` выполняется следующим образом:

- Во-первых, определяется набор достижимых элементов с именем `N`:
  - Если `T` является параметром-типом, то набор представляет собой объединение наборов достижимых элементов с именем `N` в каждом из типов, определенных как первичные или вторичные ограничения для `T` (раздел 10.1.5), наряду с набором достижимых элементов с именем `N` в классе `object`.
  - В противном случае набор содержит все достижимые элементы (раздел 3.5) с именем `N` в классе `T` и достижимые элементы с именем `N` в классе `object`. Если `T` является конструируемым типом, набор элементов получается подстановкой аргументов-типов, как описано в разделе 10.3.2. Элементы, которые содержат модификатор `override`, исключаются из набора.
- Далее, если `K` равно нулю, все вложенные типы, объявления которых содержат параметры-типы, удаляются из набора. Если `K` не равно нулю, все элементы с отличающимся числом параметров-типов удаляются. Отметим, что когда `K` равно нулю, методы, имеющие параметры-типы, не удаляются, поскольку процесс выведения типов (раздел 7.5.2) дает возможность вывести аргументы-типа.
- Далее, если элемент *вызван*, все *не-вызываемые* элементы удаляются из набора.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Это правило позволяет вызывать метод расширения, даже если в типе `T` существует не-вызываемый элемент экземпляра с таким же именем. Например, можно вызвать метод расширения `Count()` для коллекции, даже если имеется свойство экземпляра `Count` типа `int`.

- Далее из набора удаляются элементы, скрытые другими элементами. Для каждого элемента `S.M` в наборе, где `S` является типом, в котором объявлен элемент `M`, применимы следующие правила:

    - Если M является константой, полем, свойством, событием или элементом перечисления, то из набора удаляются все элементы, объявленные в базовом типе S.
    - Если M является объявлением типа, то из набора удаляются все объявления в родительском типе S, не являющиеся типами, с таким же числом параметров-типов, как у M.
    - Если M является методом, то из набора удаляются все элементы, не являющиеся методами, объявленные в родительском типе S.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Удаление из набора любого элемента на этом шаге может привести к тому, что какие-то другие элементы будут также удалены из набора. Таким образом, порядок, в котором обрабатываются элементы, не важен.

- Далее из набора удаляются интерфейсные элементы, скрытые элементами класса. Этот шаг имеет значение, только если T является параметром-типом и при этом имеет как эффективный базовый класс, отличный от класса `object`, так и непустой набор эффективных интерфейсов (раздел 10.1.5). Для каждого элемента `S.M` из набора, где S является типом, в котором объявлен элемент M, применимы следующие правила, если S является объявлением класса, отличного от `object`:
    - Если M является константой, свойством, полем, событием, элементом перечисления или объявлением типа, то все элементы, объявленные в объявлении интерфейса, удаляются из набора.
    - Если M является методом, то все элементы, не являющиеся методами, объявленные в объявлении интерфейса, удаляются из набора, и все методы с такими же сигнатурами, как у M, объявленные в объявлении интерфейса, удаляются из набора.
- Наконец, после удаления скрытых элементов результат поиска определяется следующим:
    - Если набор состоит из единственного элемента, не являющегося методом, то этот элемент и будет результатом поиска.
    - Иначе, если набор содержит только методы, то эта группа методов является результатом поиска.
    - Иначе результат поиска неоднозначен и выдается ошибка времени связывания.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Компилятор Microsoft C# более снисходителен в этом последнем случае. Если набор содержит как методы, так и элементы, методами не являющиеся, генерируется предупреждение CS0467, все элементы, не являющиеся методами, исключаются, и оставшаяся группа методов становится результатом поиска. Такое поведение оказывается полезным в некоторых сценариях, обеспечивающих взаимодействие с COM.

Для поиска элемента в типах, не являющихся параметрами-типами и интерфейсами, а также в интерфейсах, которые поддерживают только единичное наследование (каждый интерфейс в цепочке наследования имеет в точности один базовый класс или не имеет родительского класса), действие правил поиска состоит просто в том, что производные элементы скрывают базовые элементы с такими же именами или сигнатурами. Такой поиск при единичном наследовании никогда не является неоднозначным. Неоднозначность может возникнуть только при поиске элемента в интерфейсах с множественным наследованием, описанным в разделе 13.2.5.

7.4.1. Базовые типы

Для целей поиска элемента тип `T` считается имеющим следующие базовые типы:

- Если `T` есть `object`, то `T` не имеет базового типа.
- Если `T` есть *перечисление*, базовыми типами для `T` являются классы `System.Enum`, `System.ValueType` и `object`.
- Если `T` есть *структура*, базовыми типами для `T` являются классы `System.ValueType` и `object`.
- Если `T` есть *класс*, базовыми типами для `T` являются базовые классы `T`, в том числе класс `object`.
- Если `T` есть *интерфейс*, базовыми типами для `T` являются базовые интерфейсы `T` и класс `object`.

> **ЭРИК ЛИППЕРТ**
>
> Этот пункт гарантирует, что допустимо вызывать `ToString()` и другие элементы `System.Object` для значений интерфейсного типа. Во время выполнения программы любое значение интерфейсного типа будет либо нулем, либо наследником `System.Object`, так что это разумный выбор.

- Если `T` есть *тип-массива*, базовыми типами для `T` являются классы `System.Array` и `object`.
- Если `T` есть *тип-делегата*, базовыми типами для `T` являются классы `System.Delegate` и `object`.

7.5. Функциональные элементы

Функциональными элементами являются элементы, содержащие выполняемые операторы. Функциональные элементы всегда являются элементами типов и никогда не могут быть элементами пространств имен. В C# определены следующие категории функциональных элементов:

- Методы.
- Свойства.

- События.
- Индексаторы.
- Операции, определенные пользователем.
- Конструкторы экземпляра.
- Статические конструкторы.
- Деструкторы.

За исключением деструкторов и статических конструкторов (которые не могут быть вызваны явно), операторы, содержащиеся в функциональных элементах, выполняются с помощью вызова функционального элемента. Синтаксис функционального элемента зависит от его категории.

Список аргументов вызова функционального элемента (раздел 7.5.1) обеспечивает фактические значения или ссылки на переменные для параметров функционального элемента.

Вызовы обобщенных методов могут использовать выведение типов для определения набора аргументов типа, допустимых для метода. Этот процесс описан в разделе 7.5.2.

Вызовы методов, индексаторы, операции и конструкторы экземпляра используют разрешение перегрузки для определения кандидата из набора функциональных элементов, который будет вызван. Этот процесс описан в разделе 7.5.3.

Функциональный элемент идентифицируется во время связывания, возможно, через разрешение перегрузки, процесс вызова функционального элемента во время выполнения программы описан в разделе 7.5.4.

Следующая таблица подводит итог процессам обработки, происходящим в конструкциях, содержащих шесть категорий функциональных элементов, которые могут быть вызваны явно. В этой таблице `e`, `x`, `y` и `value` обозначают выражения, классифицируемые как переменные или значения, `T` обозначает выражение, классифицируемое как тип, `F` есть простое имя метода, а `P` есть простое имя свойства.

Конструкция	**Пример**	**Описание**
Вызов метода	`F(x, y)`	Разрешение перегрузки применяется для выбора наиболее подходящего метода `F` в содержащем его классе или структуре. Метод вызывается со списком аргументов `(x, y)`. Если метод не является статическим (`static`), выражение экземпляра есть `this`
	`T.F(x, y)`	Разрешение перегрузки применяется для выбоа наиболее подходящего метода в классе или структуре `T`. Ошибка времени связывания появляется, если метод не является статическим. Метод вызывается со списком аргументов `(x, y)`
	`e.F(x, y)`	Разрешение перегрузки применяется для выбора наиболее подходящего метода в классе, структуре или интерфейсе, которые определяются типом `e`. Если метод является статическим (`static`), выдается ошибка времени связывания. Метод вызывается с выражением экземпляра `e` и списком аргументов `(x, y)`

продолжение ☞

Конструкция	Пример	Описание
Доступ к свойству	`P`	Вызывается код доступа `get` к свойству в содержащем его классе или структуре. Если `P` является свойством только для записи, выдается ошибка компиляции. Если `P` не является статическим (`static`), выражение экземпляра есть `this`
	`P = value`	Вызывается код доступа `set` к свойству `P` в содержащем его классе или структуре со списком аргументов (`value`). Если `P` является свойством только для чтения, выдается ошибка компиляции. Если `P` не является статическим (`static`), выражение экземпляра есть `this`
	`T.P`	Вызывается код доступа `get` к свойству `P` в классе или структуре `T`
	`T.P = value`	Вызывается код доступа `set` к свойству `P` в классе или структуре `T` со списком аргументов (`value`). Если `P` не является статическим (`static`) или если `P` является свойством только для чтения, выдается ошибка компиляции
	`e.P`	Вызывается код доступа `get` к свойству `P` в классе, структуре или интерфейсе, которые определяются типом `e`, с выражением экземпляра `e`. Если `P` не является статическим (`static`) или если `P` является свойством только для записи, выдается ошибка компиляции
	`e.P = value`	Вызывается код доступа `set` к свойству `P` в классе, структуре или интерфейсе, которые определяются типом `e`, с выражением экземпляра `e` и списком аргументов (`value`), если `P` является статическим (`static`) или если `P` является свойством только для чтения, выдается ошибка времени связывания
Доступ к событию	`E += value`	Вызывается код доступа `add` к событию `E` в объемлющем классе или структуре. Если `E` не является статическим, выражение экземпляра есть `this`.
	`E -= value`	Вызывается код доступа `remove` к событию `E` в объемлющем классе или структуре. Если `E` не является статическим, выражение экземпляра есть `this`
	`T.E += value`	Вызывается код доступа `add` к событию `E` в классе или структуре `T`. Если `E` не является статическим, выдается ошибка времени связывания
	`T.E -= value`	Вызывается код доступа `remove` к событию `E` в классе или структуре `T`. Если `E` не является статическим, выдается ошибка времени связывания
	`e.E += value`	Вызывается код доступа `add` к событию `E` в классе, структуре или интерфейсе, которые определяются типом `e`, с выражением экземпляра `e`. Если `E` не является статическим, выдается ошибка времени связывания
	`e.E -= value`	Вызывается код доступа `remove` к событию `E` в классе, структуре или интерфейсе, которые определяются типом `e`, с выражением экземпляра `e`. Если `E` не является статическим, выдается ошибка времени связывания
Доступ к индексатору	`e[x, y]`	Разрешение перегрузки применяется для выбора наиболее подходящего индексатора в классе, структуре или интерфейсе, которые определяются типом `e`. Вызывается код доступа `get` к индексатору с выражением экземпляра `e` и списком аргументов (`x, y`). Если индексатор предназначен только для записи, выдается ошибка времени связывания

Конструкция	Пример	Описание
	`e[x, y] = value`	Разрешение перегрузки применяется для выбора наиболее подходящего индексатора в классе, структуре или интерфейсе, которые определяются типом `e`. Вызывается код доступа `set` к индексатору с выражением индексатора `e` и списком аргументов `(x, y, value)`. Если индексатор предназначен только для чтения, выдается ошибка времени связывния
Вызов операции	`-x`	Разрешение перегрузки применяется для выбора наиболее подходящей унарной операции в классе или структуре, определяемом (-ой) типом `x`. Выбранная операция вызывается со списком аргументов `(x)`
	`x + y`	Разрешение перегрузки применяется для выбора наиболее подходящей бинарной операции в классах или структурах, определяемых типами `x` и `y`. Выбранная операция вызывается со списком аргументов `(x, y)`
Вызов конструктора экземпляра	`new T(x, y)`	Разрешение перегрузки применяется для выбора наиболее подходящего конструктора экземпляра в классе или структуре T. Конструктор экземпляра вызывается со списком аргументов `(x, y)`

7.5.1. Списки аргументов

Каждый вызов функционального элемента или делегата содержит список аргументов, который обеспечивает фактические значения или ссылки на переменные для параметров функционального элемента. Синтаксис для определения списка аргументов вызова функционального элемента зависит от категории функционального элемента:

- Для конструкторов экземпляров, методов, индексаторов и делегатов аргументы определяются *списком-аргументов*, как это описано ниже. Для индексаторов, когда вызывается код доступа `set`, список аргументов дополнительно содержит выражение, определенное как правый операнд операции присваивания.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Этот добавочный аргумент не принимает участия в разрешении перегрузки для индексаторов.

- Для свойств, когда вызывается код доступа `get`, список аргументов является пустым и содержит выражение, определенное как правый операнд операции присваивания при вызове кода доступа `set`.
- Для событий список аргументов содержит выражение, определенное как правый операнд операции `+=` или `-=`.
- Для операций, определенных пользователем, список аргументов состоит из единственного операнда унарной операции или из двух операндов бинарной операции.

Аргументы свойств (раздел 10.7), событий (раздел 10.8) и операций, определенных пользователем (раздел 10.10), всегда передаются как параметры-значения (раздел 10.6.1.1). Аргументы индексаторов (раздел 10.9) всегда передаются как параметры-значения (раздел 10.6.1.1) или как параметры-массивы (раздел 10.6.1.4). Для этих категорий функциональных элементов параметры-ссылки и выходные параметры не поддерживаются.

Аргументы вызовов конструктора экземпляра, метода, индексатора или делегата определяются как *список-аргументов*:

```
список-аргументов:
   аргумент
   список-аргументов   ,   аргумент

аргумент:
   имя-аргумента_opt   значение-аргумента

имя-аргумента:
   идентификатор   :

значение-аргумента:
   выражение
   ref   ссылка-на-переменную
   out   ссылка-на-переменную
```

Список-аргументов состоит из одного или нескольких аргументов, разделенных запятыми. Каждый аргумент включает в себя не обязательное *имя-аргумента*, за которым следует *значение-аргумента*. На аргумент с именем ссылаются как на *именованный аргумент*, аргумент без имени является *позиционным аргументом*. Если позиционный аргумент появляется в списке аргументов после именованного аргумента, это является ошибкой.

Значение-аргумента может иметь одну из следующих форм:

- *Выражение*, что означает, что аргумент передается как параметр-значение (раздел 10.6.1.1).
- Ключевое слово `ref`, за которым следует *ссылка-на-переменную* (раздел 5.4); это означает, что аргумент передается как параметр-ссылка (раздел 10.6.1.2). Переменная должна быть явно присвоенной (раздел 5.3), прежде чем она может быть передана в качестве параметра-ссылки.
- Ключевое слово `out`, за которым следует *ссылка-на-переменную* (раздел 5.4); это означает, что аргумент передается как выходной параметр (раздел 10.6.1.3). Переменная считается явно присвоенной после вызова функционального элемента, в который эта переменная передается как выходной параметр.

7.5.1.1. Соответствующие параметры

Для каждого аргумента в списке аргументов в вызываемом функциональном элементе или делегате должен быть соответствующий параметр.

Список параметров, использующийся в дальнейшем изложении, определяется следующим образом:

- Для виртуальных методов и индексаторов, определенных в классах, список параметров выбирается из наиболее точного объявления или переопределения функционального элемента; поиск начинается со статического типа получателя и продолжается в его базовых классах.

> **ЭРИК ЛИППЕРТ**
>
> Это правило предназначено для неудачной (и, будем надеяться, маловероятной) ситуации, когда у вас есть виртуальный метод, скажем, `void M(int x, int y)`, и подмененный метод, скажем, `void M(int y, int x)`. Будет ли при вызове метода `M(y:1, x:2)` в действительности вызван метод `M(1, 2)` или метод `M(2, 1)`? Параметры не являются частью сигнатуры метода и, таким образом, могут изменяться при перегрузке.

- Для интерфейсных методов и индексаторов список параметров выбирается из наиболее точного определения элемента; поиск начинается с типа интерфейса и продолжается в его базовых интерфейсах. Если единственный список параметров не найден, создается список параметров с недостижимыми именами и не имеющий необязательных параметров, так что при вызове невозможно использовать именованные параметры или пропустить необязательные аргументы.
- Для частичных методов используется список параметров из объявления частичного метода.
- Для всех остальных функциональных элементов и делегатов существует единственный список параметров, который и используется.

Позиция аргумента или параметра определяется числом аргументов или параметров, предшествующих ему в списке аргументов или в списке параметров.

Соответствующие параметры для аргументов функциональных элементов устанавливаются следующим образом:

- Аргументы в *списке-аргументов* конструкторов экземпляра, методов, индексаторов и делегатов:
  - Позиционный аргумент, фиксированный параметр которого находится в той же позиции в списке параметров, соответствует этому параметру.
  - Позиционный аргумент функционального элемента с параметром-массивом, вызванного в его нормальной форме, соответствует параметру-массиву, который должен находиться в той же позиции в списке параметров.
  - Позиционный аргумент функционального элемента с параметром-массивом, вызванного в его расширенной форме, где никакой фиксированный параметр не появляется в той же позиции в списке параметров, соответствует элементу параметра-массива.
  - Именованный аргумент соответствует параметру с тем же именем в списке параметров.
  - Для индексаторов, когда вызывается код доступа `set`, выражение, определенное как правый операнд операции присваиванияния, соответствует неявному параметру `value` в объявлении кода доступа `set`.

- Для свойств, когда вызывается код доступа `get`, аргументов не существует. Когда вызывается код доступа `set`, выражение, определенное как правый операнд оператора присваивания, соответствует неявному параметру `value` в объявлении кода доступа `set`.
- Для определенных пользователем унарных операций (включая преобразования) единственный операнд соответствует единственному параметру в объявлении операции.
- Для определенных пользователем бинарных операций левый операнд соответствует первому параметру, а правый операнд соответствует второму параметру в объявлении операции.

7.5.1.2. Вычисление списка аргументов во время выполнения программы

При обработке вызова функционального элемента во время выполнения программы (раздел 7.5.4) выражения или ссылки на переменные списка аргументов вычисляются по порядку слева направо следующим образом:

- Для параметра-значения вычисляется выражение аргумента и выполняется неявное преобразование (раздел 6.1) к соответствующему типу параметра. Результирующее значение становится начальным значением параметра-значения при вызове функционального элемента.
- Для параметра-ссылки или выходного параметра вычисляется ссылка на переменную, и на полученную в результате ячейку хранения будет ссылаться параметр при вызове функционального элемента. Если ссылка на переменную, заданная в качестве выходного параметра, является элементом массива *ссылочного-типа*, выполняется проверка во время выполнения программы, чтобы убедиться, что тип элементов массива идентичен типу параметра. Если эта проверка не проходит успешно, генерируется исключение `System.ArrayTypeMismatchException`.

В методах, индексаторах и конструкторах экземпляра крайний справа параметр можеть объявляться как параметр-массив (раздел 10.6.1.4). Такие функциональные элементы вызываются либо в нормальной, либо в расширенной форме в зависимости от того, какая из них применима (раздел 7.5.3.1):

- Когда функциональный элемент с параметром-массивом вызывается в нормальной форме, аргумент, заданный для этого параметра-массива, должен быть одиночным выражением, неявно преобразуемым (раздел 6.1) к типу параметра-массива. В данном случае параметр-массив ведет себя в точности как параметр-значение.
- Когда функциональный элемент с параметром-массивом вызывается в расширенной форме, вызов должен определять ноль или более позиционных аргументов для параметра-массива, где каждый аргумент есть выражение, неявно преобразуемое (раздел 6.1) к типу элементов параметра-массива. В этом случае при вызове создается экземпляр параметра-массива с длиной, соответствующей

количеству аргументов, элементы экземпляра массива инициализируются данными значениями аргументов, и вновь созданный экземпляр массива используется в качестве фактического аргумента.

Выражения в списке аргументов всегда вычисляются в том порядке, в котором они записаны. Таким образом, пример:

```
class Test
{
   static void F(int x, int y = -1, int z = -2)
   {
      System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
   }
   static void Main()
   {
      int i = 0;
      F(i++, i++, i++);
      F(z: i++, x: i++);
   }
}
```

выведет:

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

Правила ковариантности для массивов (раздел 12.5) позволяют значению типа массива `A[]` быть ссылкой на экземпляр типа массива `B[]`, при условии, что существует неявное ссылочное преобразование из `B` к `A`. Вследствие этих правил в случае, когда элемент массива *ссылочного-типа* передается как параметр-ссылка или выходной параметр, во время выполнения требуется проверка, чтобы убедиться, что фактический тип элемента массива *идентичен* типу параметра. В примере

```
class Test
{
   static void F(ref object x) {...}
   static void Main()
   {
      object[] a = new object[10];
      object[] b = new string[10];
      F(ref a[0]);              // Правильно
      F(ref b[1]);              // Инсключение ArrayTypeMismatchException
   }
}
```

второй вызов `F` порождает исключение `System.ArrayTypeMismatchException`, поскольку фактический тип `b` есть `string`, а не `object`.

Когда функциональный элемент с параметром-массивом вызывается в расширенной форме, вызов обрабатывается в точности так, как если бы выражение создания массива с инициализатором массива (раздел 7.6.10.4) находилось между расширенными параметрами. Например, пусть дано объявление:

```
void F(int x, int y, params object[] args);
```

Следующие вызовы метода в расширенной форме:

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

в точности соответствуют

```
F(10, 20, new object[] { });
F(10, 20, new object[] { 30, 40 });
F(10, 20, new object[] { 1, "hello", 3.0 });
```

В частности, отметим, что когда для параметра-массива задано ноль аргументов, создается пустой массив.

Когда аргументы функционального элемента с соответствующими необязательными параметрами опускаются, неявно передаются аргументы по умолчанию, заданные в объявлении функционального элемента. Поскольку они всегда являются константами, их вычисление не влияет на порядок вычисления оставшихся аргументов.

7.5.2. Выведение типов

Когда выполняется вызов обобщенного метода без указания аргументов-типов, запускается процесс **выведения типов**, задача которого — вывести типы аргументов для вызова. Наличие такого процесса дает возможность использовать более удобный синтаксис вызова обобщенных методов и позволяет программисту не указывать избыточную информацию о типах. Например, данное объявление метода:

```
class Chooser
   {
   static Random rand = new Random();
   public static T Choose<T>(T first, T second)
   {
      return (rand.Next(2) == 0) ? first : second;
   }
}
```

позволяет вызвать метод `Choose` без точного определения аргументов-типов:

```
int i = Chooser.Choose(5, 213);              // Вызывается Choose<int>
string s = Chooser.Choose("foo", "bar");     // Вызывается Choose<string>
```

С помощью выведения типов аргументы-типы `int` и `string` определяются из аргументов для этого метода.

> **ДЖОН СКИТ**
>
> Однажды я пытался разобраться в алгоритме выведения для особых случаев, когда происходило не то, чего я ожидал. Я почти полностью разочаровался в нем, выйдя из этого процесса в гораздо большем замешательстве, чем был до него. Чувствуешь себя как будто в лабиринте, надеешься найти сокровище в центре (подходящие аргументы-типы), но обычно по дороге встречаешь Минотавра. Я испытываю благоговейный трепет перед теми, кто не только сориентировался в этом алгоритме, но еще и воплотил его в компиляторе.

> **КРИС СЕЛЛЗ**
>
> Если понимаешь, что твой код дает возможность почувствовать себя в лабиринте, значит, что-то сделано неправильно.

Выведение типа выполняется как часть обработки вызова метода во время связывания (раздел 7.6.5.1) и происходит перед этапом разрешения перегрузки. Когда при вызове метода определена некоторая группа методов и никакие аргументы типа не определены как часть вызова метода, к каждому обобщенному методу из группы методов применяется выведение типов. Если выведение типов проходит успешно, то выведенные аргументы-типы используются для определения типов аргументов для последующего разрешения перегрузки. Если в результате разрешения перегрузки выбирается единственный обобщенный метод, который должен быть вызван, то аргументы выведенного типа используются как фактические аргументы-типы во время вызова. Если выведение типа для определенного метода завершилось неудачно, этот метод не будет принимать участие в разрешении перегрузки. Само по себе неудачное завершение выведения типов не является причиной ошибки времени связывания. Однако оно часто приводит к ошибке времени связывания, когда затем разрешение перегрузки не позволяет найти какие-либо подходящие методы.

ЭРИК ЛИППЕРТ

Алгоритм выведения типов не гарантирует нахождение подходящего кандидата. (Конечно, разрешение перегрузки приводит к удалению неподходящих кандидатов.) Алгоритм выведения типов создан для ответа на один вопрос: какой *наилучший возможный* аргумент-тип может быть выведен для каждого параметра-типа, если заданы только аргументы и типы формальных параметров? Если наилучшее возможное выведение дает в результате неподходящего кандидата, мы не можем проследить ход мысли пользователя и отгадать, что же он «на самом деле» имел в виду такое, чтобы результат выведения мог бы быть другим.

МАРЕК САФАР

Когда выведение типов проходит успешно, а разрешение перегрузки — нет, виновником может быть выведенный параметр типа, имеющий ограничения. Алгоритм выведения типов игнорирует ограничения параметров-типов, оставляя разрешению перегрузки проверку наилучшего кандидата.

Если количество имеющихся аргументов-типов отличается от количества параметров метода, выведение типов немедленно завершается неудачно. Предположим, что обобщенный метод имеет следующую сигнатуру:

$$Tr\ M<X_1...X_n>(T_1\ x_1\ ...\ T_m\ x_m)$$

Для вызова метода в форме $M(E_1 ... E_m)$ задачей выведения типов является нахождение уникальных аргументов-типов $S_1...S_n$ для каждого из параметров-типов $X_1...X_n$, таких, чтобы вызов метода $M<S_1...S_n>(E_1...E_m)$ работал.

ВЛАДИМИР РЕШЕТНИКОВ

Хотя здесь про это явно не говорится, сигнатура может иметь параметры `ref/out`.

В течение процесса выведения каждый параметр X_i является либо *фиксированным* на определенном типе S_i, либо *нефиксированным* со связанным с ним набором *границ*.

Выведение типов происходит по фазам. В каждой фазе делается попытка вывести аргументы-типы для большего числа переменных типа, основываясь на том, что уже найдено во время предыдущей фазы. В первой фазе выполняется некоторое начальное выведение границ, во второй фазе переменные типов фиксируются на определенных типах и происходит дальнейшее выведение границ. Вторая фаза может повторяться несколько раз.

Примечание. Выведение типов выполняется не только при вызове обобщенного метода. Выведение типов для преобразования групп методов описано в разделе 7.5.2.13, а нахождение наилучшего общего типа для набора выражений описано в разделе 7.5.2.14.

ЭРИК ЛИППЕРТ

У команды разработчиков языка иногда спрашивают: «Зачем вы изобретали собственный алгоритм выведения типов, вместо того чтобы использовать алгоритм Хиндли—Милнера (Hindley–Milner-style algorithm)?». Короткий ответ:
(1) Алгоритм Хиндли—Милнера с трудом адаптируем к языкам с наследованием, основанным на классах, и ковариантными типами.
(2) Алгоритмы поиска с возвратом могут выполняться очень много времени.
Наш алгоритм выведения типов сконструирован так, чтобы удобно было иметь дело с наследованием и вариантностью типов. Поскольку каждая итерация во второй фазе либо «фиксирует» добавляемый параметр-тип на его выведенном типе, либо терпит при этом неудачу, ясно, что процесс выведения типов заканчивается в максимально короткий срок.

ПИТЕР СЕСТОФТ

Хотя, возможно, и ненамеренно, предыдущее замечание оставило впечатление, что алгоритм выведения типов для системы типов Хиндли—Милнера (эта система допускает полиморфизм) должен использовать поиск с возвратом. В частности, алгоритм Дамаса—Милнера (Damas–Milner algorithm) (1978) не использует поиск с возвратом. Поиск с возвратом, так сказать, опрокинул бы все выведение типов в C#, поскольку в C# все параметры типов должны быть явно определены. Напротив, алгоритм Дамаса—Милнера будет изобретать столько параметров-типов, чтобы найти наиболее общий тип для данного выражения. Известно, что время решения этой задачи растет по экспоненте.

7.5.2.1. Первая фаза

Для каждого аргумента метода E_i:

- Если E_i является анонимной функцией, *явное выведение типа параметра* (раздел 7.5.2.7) выполняется *от* E_i *к* T_i.
- Иначе, если E_i имеет тип U и x_i является параметром-значением, то *выведение нижней границы* (*lower-bound inference*) выполняется от U к T_i.

- Иначе если E_i имеет тип U и x_i является параметром `ref` или `out`, то точное выведение выполняется от U to T_i.
- Иначе для данного аргумента выведение не выполняется.

7.5.2.2. Вторая фаза

Вторая фаза выполняется следующим образом:

- Все *нефиксированные* переменные типа X_i, которые *не зависят* от любых X_j, *фиксируются* (раздел 7.5.2.10).
- Если не существует таких переменных типа, *фиксируются* все *нефиксированные* переменные типа X_i, для которых выполняются оба следующих ниже условия:
  - Имеется по меньшей мере одна переменная типа X_j, которая *зависит от* X_i.
  - X_i имеет непустой набор границ.
- Если не существует таких переменных типа и все еще существуют *нефиксированные* переменные типа, выведение типов завершается неудачно.
- В противном случае, если более не существует *нефиксированных* переменных типа, выведение завершается успешно.
- Иначе для всех аргументов E_i с соответствующим типом параметра T_i, для которого *выходные типы* содержат нефиксированные переменные типа X_j, а *входные типы* не содержат таковых, *выведение выходного типа* (раздел 7.5.2.6) выполняется *от* E_i *к* T_i. Затем вторая фаза повторяется.

7.5.2.3. Входные типы

Если E есть группа методов или анонимная функция с неявно заданным типом и T есть тип делегата или тип дерева выражений, то все типы параметров T являются *входными типами* E *с типом* T.

7.5.2.4. Выходные типы

Если E есть группа методов или анонимная функция с неявно заданным типом и T есть тип делегата или тип дерева выражений, то тип, возвращаемый T, является выходным типом E *с типом* T.

7.5.2.5. Зависимость

Нефиксированная переменная типа X_i *прямо зависит* от *нефиксированной* переменной типа X_j, если для некоторого аргумента с типом T_k X_j появляется во *входном типе* E_k с типом T_k и X_i появляется в *выходном типе* E_k с типом T_k.

X_j зависит от X_i, если X_j прямо зависит от X_i, или если X_i прямо зависит от X_k и X_k зависит от X_j. Таким образом, «зависит» транзитивно, но не рефлексивно замкнуто по отношению к «зависит прямо».

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Параметр-тип может прямо или косвенно зависеть сам от себя.

7.5.2.6. Выведение выходного типа

Выведение выходного типа выполняется *от* выражения E *к* типу T следующим образом:

- Если E есть анонимная функция с выведенным возвращаемым типом U (раздел 7.5.2.12) и T есть тип делегата или тип дерева выражений с возвращаемым типом T_b, то *выведение нижней границы* выполняется *от* U *к* T_b.
- Иначе, если E есть группа методов и T есть тип делегата или тип дерева выражений с типами параметров $T_1...T_k$ и типом возвращаемого значения T_b и разрешение перегрузки с типами $T_1...T_k$ дает единственный метод с типом возвращаемого значения U, то *выведение нижней границы* выполняется *от* U *к* T_b.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Этот шаг выполняется, только если все параметры типа метода, встречающиеся в типах параметра делегата, уже фиксированы. Разрешение перегрузки не пытается выбрать наилучший метод, основываясь на неполной информации о типах.

- Иначе, если E есть выражение с типом U, то выведение нижней границы выполняется от U к T.
- Иначе выведение типов не выполняется.

7.5.2.7. Явное выведение типа параметра

Явное выведение типа параметра выполняется *от* выражения E *к* типу T следующим образом:

- Если E является анонимной функцией с типами параметров $U_1...U_k$ и T является типом делегата или типом дерева выражений с типами параметров $V_1...V_k$, то для каждого U_i *точное выведение* (раздел 7.5.2.8) выполняется от U_i к соответствующему V_i.

7.5.2.8. Точное выведение

Точное выведение от типа U к типу V выполняется следующим образом:

- Если V является одним из типов *нефиксированных* переменных X_i, то U добавляется к набору точных границ X_i.
- Иначе наборы $V_1...V_k$ и $U_1...U_k$ определяются проверкой на наличие одного из следующих вариантов:

- V является типом массива $V_1[...]$ и U является типом массива $U_1[...]$, и массивы V и U имеют одинаковую размерность.
- V является типом $V_1?$ и U является типом $U_1?$.
- V является сконструированным типом $C<V_1...V_k>$ и U является сконструированным типом $C<U_1...U_k>$.
- Если имеет место один из этих вариантов, то *точное выведение* выполняется *от* каждого U_i *к* соответствующему V_i.

- Иначе выведение не выполняется.

7.5.2.9. Выведение нижней границы

Выведение нижней границы от типа U к типу V выполняется следующим образом:

- Если V является одним из типов *нефиксированных* переменных X_i, то U добавляется к набору нижних границ X_i.
- Иначе наборы $V_1...V_k$ и $U_1...U_k$ определяются проверкой на наличие одного из следующих вариантов:
    - V является типом массива $V_1[...]$ и U является типом массива $U_1[...]$ (или параметром типа, эффективным базовым типом которого является $U_1[...]$), и массивы V и U имеют одинаковую размерность (число размерностей).
    - V является одним из типов $IEnumerable<V_1>$, $ICollection<V_1>$ или $IList<V_1>$, и U является одномерным массивом типа $U_1[]$ (или параметром типа, эффективный базовый тип которого есть $U_1[]$).

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Конечно, в этом списке (и в соответствующем списке следующего раздела) упоминаются интерфейсы из пространства имен `System.Collections.Generic`.

- V является типом $V_1?$ и U является типом $U_1?$.
- V является сконструированным классом, структурой, интерфейсом или делегатом типа $C<V_1...V_k>$, и имеется единственный тип $C<U_1...U_k>$, такой что U (или, если U является параметром-типом, его эффективный базовый класс или любой элемент из набора его эффективных интерфейсов) тождественен $C<U_1...U_k>$, наследует от него (прямо или косвенно) или реализует его (прямо или косвенно).

(Ограничение, связанное с «единственностью», означает, что в случае `interface C<T>{} class U: C<X>, C<Y>{}` при выведении от U к `C<T>` выведение не выполняется, поскольку U_1 может быть X или Y.)

Если имеет место один из таких случаев, то выведение выполняется от каждого U_i к соответствующему V_i следующим образом:

- Если известно, что U_i не является ссылочным типом, то выполняется *точное выведение*.

- Иначе, если V является типом массива, то выполняется *выведение нижней границы.*
- Иначе, если U есть `C<U1...Uk>`, то выведение зависит от `i`-го параметра типа `C`:
  - Если он является ковариантным, то выполняется *выведение нижней границы.*
  - Если он является контравариантным, то выполняется *выведение верхней границы.*
  - Если он является инвариантным, то выполняется *точное выведение.*
- Иначе выведение не выполняется.

7.5.2.10. Выведение верхней границы

Выведение верхней границы от типа U к типу V выполняется следующим образом:

- Если V является одним из типов *нефиксированных* переменных X_i, то U добавляется к набору верхних границ X_i.
- Иначе наборы V_1...V_k и U_1...U_k определяются проверкой на наличие одного из следующих вариантов:
  - U является типом массива `U1[...]` и V является типом массива `V1[...]`, и массивы V и U имеют одинаковую размерность.
  - U является одним из типов `IEnumerable<Ue>`, `ICollection<Ue>` или `IList<Ue>`, и V является одномерным массивом типа `Ve[]`.
  - U является типом `U1?` и V является типом `V1?`.
  - U является сконструированным классом, структурой, интерфейсом или делегатом типа `C<U1...Uk>`, а V является классом, структурой, интерфейсом или делегатом, тип которого идентичен единственному типу `C<U1...Uk>`, наследует от него (прямо или косвенно) или реализует его (прямо или косвенно).

(Ограничение, связанное с «единственностью», означает, что если у нас есть `interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}`, при выведении от `C<U1>` к `V<Q>` выведение не выполняется. Выведения не выполняются от U_1 к `X<Q>` или к `Y<Q>`.)

Если имеет место один из таких случаев, то выведение выполняется от каждого U_i к соответствующему V_i следующим образом:

- Если известно, что U_i не является ссылочным типом, то выполняется *точное выведение.*
- Иначе, если V является типом массива, то выполняется *выведение верхней границы.*
- Иначе, если U есть `C<U1...Uk>`, то выведение зависит от `i`-го параметра типа `C`:
  - Если он является ковариантным, то выполняется *выведение верхней границы.*
  - Если он является контравариантным, то выполняется *выведение нижней границы.*
  - Если он является инвариантным, то выполняется *точное выведение.*
- Иначе выведение не выполняется.

ВЛАДИМИР РЕШЕТНИКОВ

Таким образом, никакое выведение не может выполняться для параметров-типов метода, которые не появляются в типах параметров обобщенного метода. Например, выведение не выплоняется для параметров, которые появляются только в ограничениях:

```
using System.Collections.Generic;

class C
{
   static void Foo<T,U>(T x) where T : IEnumerable<U> { }
   static void Main()
   {
      Foo(new List<int>());
      // Ошибка CS0411: The type arguments for method ‘C.Foo<T,U>(T)’
      // cannot be inferred from the usage
      // (невозможно вывести аргументы-типы для метода)
   }
}
```

7.5.2.11. Фиксация

Нефиксированная переменная типа X_i с набором границ *фиксируется* следующим образом:

- Набор *возможных типов* U_j первоначально устанавливается как набор всех типов, заключенных в рамках границ для X_i.

ЭРИК ЛИППЕРТ

Это условие иллюстрирует одну из тонкостей C#. Когда возникает необходимость выбрать «наилучший тип» из данного набора типов, мы всегда выбираем один из типов в наборе. То есть если стоит задача выбрать наилучший тип из набора {`Кошка, Собака, Золотая рыбка`}, у нас ничего не получится; мы же не можем сказать: «Общий базовый тип `Животные` есть лучший из этих трех типов» —потому, что тип `Животные` не содержится в данном наборе.

- Затем поочередно проверяется каждая граница для X_i: для каждой границы `U` для X_i все типы U_j, которые не тождественны `U`, удаляются из набора кандидатов. Для каждой нижней границы `U` для X_i все типы U_j, к которым *нет* неявного приведения от `U`, удаляются из набора кандидатов. Для каждой верхней границы `U` для X_i все типы U_j, от которых *нет* неявного приведения к `U`, удаляются из набора кандидатов.
- Если среди оставшихся возможных типов U_j есть уникальный тип `V`, от которого существует неявное приведение ко всем остальным возможным типам, то X_i фиксируется на `V`.
- В противном случае выведение типов завершается неудачно.

> **ЭРИК ЛИППЕРТ**
>
> В C# 2.0 алгоритм выведения типа обобщенного метода завершался неудачно, если для одного и того же параметра-типа были вычислены две отдельные границы. В C# 3.0 мы сталкиваемся с ситуациями, такими как в случае метода `Join`, когда нужно осуществить выведение типа «ключа» для двух объединяемых коллекций. Если ключ для одной из коллекций будет, скажем, `int`, а для другой коллекции соответствующий ключ будет `int?`, то мы имеем два отдельных типа. Поскольку любой `int` может быть приведен к `int?`, мы можем допустить неоднозначность и разрешить ее, выбрав более общий из двух типов. C# 4.0 поддерживает обобщенную вариантность, которая еще больше усложняет ситуацию; теперь мы имеем верхнюю, нижнюю и точную границы параметра-типа.

7.5.2.12. Выведенный тип возвращаемого значения

Выведенный тип возвращаемого значения анонимной функции `F` используется при выведении типа и разрешении перегрузки. Выведенный тип возвращаемого значения может быть определен только для анонимной функции, у которой известны все параметры-типы: либо они заданы в явном виде через приведение анонимной функции, либо выведены в процессе выведения типов при вызове содержащего ее обобщенного метода. Выведеный тип возвращаемого значения определяется следующим образом:

- Если тело `F` является *выражением*, то выведенным типом возвращаемого значения `F` является тип этого выражения.
- Если тело `F` является *блоком* и набор выражений в операторах блока `return` имеет наилучший общий тип `T` (раздел 7.5.2.14), то выведенный возвращаемый тип `F` есть `T`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Даже операторы `return` в недостижимом коде включены в набор и принимают участие в вычислении выведенного возвращаемого типа. Это имеет то удивительное следствие, что удаление недостижимого кода может изменять поведение программы.

- Иначе тип возвращаемого значения для `E` не может быть выведен.

В качестве примера выведения типа с участием анонимной функции рассмотрим метод расширения `Select`, объявленный в классе `System.Linq.Enumerable`:

```
namespace System.Linq
{
   public static class Enumerable
   {
      public static IEnumerable<TResult> Select<TSource, TResult>(
         this IEnumerable<TSource> source,
         Func<TSource, TResult> selector)
```

```
      {
         foreach (TSource element in source)
         yield return selector(element);
      }
   }
}
```

Предположим, что пространство имен System.Linq было импортировано с помощью using и существует класс Customer со свойством Name типа string, тогда метод Select можно использовать для выбора имен из списка заказчиков:

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

Вызов расширяемого метода (раздел 7.6.5.2) Select обрабатывается преобразованием вызова к вызову статического метода:

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

Так как аргументы-типы не определены явно, для аргументов-типов используется выведение типов. Во-первых, аргумент customers соотносится с параметром source, выводя тип T к типу Customer. Затем, если вспомнить процесс выведения типов для анонимной функции, описанный выше, c является заданным типом Customer и выражение c.Name соотносится с типом возвращаемого значения параметра selector, выводя S к типу string. Таким образом, этот вызов эквивалентен

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

и типом результата является IEnumerable<string>.

Следующий пример демонстрирует, как выведение типа анонимной функции позволяет информации о типе «перетекать» между аргументами при вызове обобщенного метода. Для данного метода

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
   return f2(f1(value));
}
```

выведение типа для вызова

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

выполняется следующим образом. Во-первых, аргумент «1:15:30» соотносится с параметром value, выводя X к типу string. Затем, параметру первой анонимной функции, s, задается выведенный тип string, и выражение TimeSpan.Parse(s) соотносится с возвращаемым типом f1, выводя Y к типу System.TimeSpan. Наконец, параметру второй анонимной функции, t, задается выведенный тип System.TimeSpan, и выражение t.TotalSeconds соотносится с возвращаемым типом f2, выводя Z к double. Таким образом, результат вызова имеет тип double.

7.5.2.13. Выведение типа для преобразования группы методов

Наряду с вызовами обобщенных методов выведение типов должно быть также применимо в том случае, когда группа методов M, содержащая обобщенный метод, приводится к заданному типу делегата D (раздел 6.6). Задан метод

$T_r\ M\langle X_1 \ldots X_n\rangle(T_1\ x_1\ \ldots\ T_m\ x_m)$

и группа методов M, которая присваивается делегату типа D; задачей выведения типов в данном случае является нахождение аргументов типа $S_1...S_n$, таких, что выражение

```
M<S1...Sn>
```

будет совместимым с D (раздел 15.1).

В отличие от алгоритма выведения типа для вызова обобщенного метода, в данном случае имеются только *типы* аргументов — нет *выражений* аргументов. В частности, нет анонимных функций, а следовательно, отсутствует необходимость во множественных фазах выведения.

Вместо этого, все X_i считаются *нефиксированными*, и *выведение нижней границы* выполняется от каждого типа аргумента U_j делегата D к соответствующему типу параметра T_j группы методов M. Если для какого-либо X_i границы не будут найдены, выведение типов завершится неудачно. В противном случае все X_i фиксируются на соответствующие S_i, которые и будут результатом выведения.

7.5.2.14. Нахождение наилучшего общего типа для набора выражений

В некоторых случаях общий тип необходимо вывести для набора выражений. В частности, таким способом находятся типы элементов массивов с неявно заданными типами и возвращаемые типы анонимных функций с телом в виде *блока*.

Интуитивно понятно, что для заданного набора выражений $E_1...E_m$ такое выведение должно быть эквивалентно вызову метода

```
Tr M<X>(X x1 ... X xm)
```

с аргументами E_i.

Точнее говоря, выведение начинается с *нефиксированной* переменной типа X, *выведение выходного типа,* затем выполняется от каждого E_i к X. Окончательно X *фиксируется*, и в случае успешности, тип результата S будет результирующим наилучшим общим типом для выражений. Если такого S не существует, выражения не имеют наилучшего общего типа.

7.5.3. Разрешение перегрузки

Разрешение перегрузки представляет собой механизм времени связывания для выбора наилучшего функционального элемента из набора возможных функциональных элементов для вызова при заданном списке аргументов. Разрешение перегрузки выбирает функциональный элемент, который будет вызван, в следующих контекстах языка C#:

- Вызов метода, заданного в *выражении-вызова* (раздел 7.6.5.1).
- Вызов конструктора экземпляра, заданного в *выражении-создания-объекта* (раздел 7.6.10.1).
- Вызов кода доступа к экземпляру через *доступ-к-элементу* (раздел 7.6.6).
- Вызов предопределенной или определенной пользователем операции через выражение (разделы 7.3.3 и 7.3.4)

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Такие же правила управляют выбором конструктора экземпляра атрибута в спецификации атрибута, вызовом конструктора экземпляра в *инициализаторе конструктора* и вызовом индексатора через доступ к базовому классу (*base-access*).

Каждый из этих контекстов определяет набор возможных функциональных элементов и список аргументов своим собственным уникальным способом, как детально описано в приведенных выше разделах. Например, набор кандидатов для вызова метода не включает методы, помеченные как `override` (раздел 7.4), и методы родительского класса не являются кандидатами, если какой-либо метод производного класса является подходящим (раздел 7.6.5.1).

После того как возможные функциональные элементы и список аргументов определены, выбор лучшего функционального элемента происходит одинаково во всех случаях:

- Для заданного набора подходящих возможных функциональных элементов определяется лучший функциональный элемент. Если набор содержит только один функциональный элемент, он и есть лучший. В противном случае лучшим функциональным элементом является тот, который лучше остальных соответствует данному списку аргументов, что выполняется путем сравнения каждого функционального элемента со всеми другими функциональными элементами согласно правилам раздела 7.5.3.2. Если не находится в точности один лучший функциональный элемент, вызов функционального элемента является неоднозначным и выдается ошибка времени связывания.

В следующих разделах даны точные определения значению выражений **подходящий функциональный элемент** (*applicable function member*) и **лучший функциональный элемент** (*better function member*).

> **БИЛЛ ВАГНЕР**
>
> Этот раздел получился очень сложным. Когда вы читаете его, помните, что каждая добавленная вами перегрузка может внести свой вклад в возможную неоднозначность разрешения перегрузки. Желательно ограничить число перегрузок, что облегчит использование класса. Это позволяет сделать все как можно проще для пользователей вашего класса и помогает им обрести уверенность в том, что они используют правильный метод.

7.5.3.1. Подходящий функциональный элемент

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Этот раздел также определяет правила для подходящих делегатов (раздел 7.6.5.3).

Функциональный элемент называется **подходящим функциональным элементом** для списка аргументов `A`, когда выполняются все пункты следующего списка:

- Каждый аргумент в `A` соответствует параметру в объявлении функционального элемента, как описано в разделе 7.5.11, и любой параметр, для которого нет соответствующего аргумента, является необязательным параметром.
- Для каждого аргумента в `A` способ передачи аргумента параметру (то есть `value`, `ref` или `out`) идентичен способу передачи параметра соответствующего параметра, при этом:
  - для параметра-значения или параметра-массива существует неявное приведение (раздел 6.1) от аргумента к типу соответствующего параметра, или
  - для параметров `ref` или `out` тип аргумента тождественен типу соответствующего параметра. Фактически параметры `ref` или `out` являются псевдонимами переданных аргументов.

Если функциональный элемент, который содержит параметр-массив, является подходящим в соответствии с приведенными выше правилами, то говорят, что он является подходящим **в нормальной форме**. Если функциональный элемент, который содержит параметр-массив, не является подходящим в нормальной форме, он может быть подходящим **в расширенной форме**:

- Расширенная форма строится при помощи замены параметра-массива в объявлении функционального элемента нулем или более значимых параметров типа элементов параметра-массива так, что количество аргументов в списке аргументов `A` соответствует общему количеству параметров. Если `A` имеет меньшее количество аргументов, чем количество фиксированных параметров в объявлении функционального элемента, расширенная форма функционального элемента не может быть построена и, таким образом, не может быть подходящей.
- В противном случае расширенная форма является подходящей, если для каждого аргумента в списке `A` способ передачи параметру аргумента идентичен способу передачи параметра для соответствующего параметра, и кроме того:
  - для фиксированного параметра-значения или параметра-значения, созданного при расширении, существует неявное приведение (раздел 6.1) из типа аргумента к типу соответствующего параметра, или
  - для параметров `ref` или `out` тип аргумента тождественен типу соответствующего параметра.

7.5.3.2. Лучший функциональный элемент

Для определения лучшего функционального элемента создается упрощенный список аргументов `A`, который содержит только выражения аргументов в порядке, в котором они появляются в исходном списке аргументов.

Списки параметров для каждого возможного функционального элемента создаются следующим образом:

- Расширенная форма используется, если функциональный элемент является подходящим только в расширенной форме.
- Необязательные параметры без соответствующих аргументов удаляются из списка параметров.

- Порядок расположения параметров меняется таким образом, что каждый из них появляется в той же позиции, что и соответствующий аргумент в списке аргументов.

Если задан список аргументов `A` с набором выражений аргументов { E_1, E_2, …, E_N } и два функциональных элемента M_P и M_Q с типами параметров { P_1, P_2, …, P_N } и { Q_1, Q_2, …, Q_N }, M_P определяется как *лучший функциональный элемент* по сравнению с M_Q, если выполняются следующие условия:

- Для каждого аргумента неявное приведение от E_X к Q_X не лучше, чем неявное приведение от от E_X к P_X.
- По меньшей мере для одного аргумента приведение от E_X к P_X лучше, чем приведение от E_X к Q_X.

Когда выполняется этот алгоритм, если M_P или M_Q являются подходящими в расширенной форме, то P_X или Q_X ссылаются на параметр в расширенной форме списка параметров.

Если последовательности типов параметров {P_1, P_2, …, P_N} и {Q_1, Q_2, …, Q_N} эквивалентны (то есть для каждого P_i существует тождественное преобразование к соответствующему Q_i), применимы следующие правила для разрушения связи с целью определения лучшего функционального элемента:

- Если M_P является необобщенным методом, а M_Q является обобщенным методом, то M_P лучше, чем M_Q.
- Иначе, если M_P является подходящим в его нормальной форме, а M_Q имеет массив `params` и является подходящим только в расширенной форме, то M_P лучше, чем M_Q.
- Иначе, если M_P имеет больше объявленных параметров, чем M_Q, то M_P лучше, чем M_Q. Это правило применяется, если оба метода имеют массивы `params` и являются подходящими только в их расширенной форме.
- Иначе, если все параметры M_P имеют соответствующие аргументы, в то время как по меньшей мере одному необязательному параметру в M_Q должен быть поставлен в соответствие аргумент по умолчанию, то M_P лучше, чем M_Q.
- Иначе, если M_P имеет более определенные типы параметров, чем M_Q, то M_P лучше, чем M_Q. Пусть {R_1, R_2, ..., R_N} и {S_1, S_2, ..., S_N} представляют не инстанцированные и не расширенные типы параметров M_P и M_Q. Типы параметров M_P являются более определенными, чем типы параметров M_Q, если для любого параметра R_X не менее определен, чем S_X, и по меньшей мере для одного параметра тип R_X более определен, чем S_X:
  - Параметр-тип менее определен, чем параметр, не являющийся параметром-типом.
  - Рекурсивно, сконструированный тип более определен, чем другой сконструированный тип (с таким же числом аргументов типа), если по меньшей мере один аргумент типа более определен и никакой аргумент типа не определен менее, чем соответствующий аргумент в другом типе.

- Тип массива более определен, чем другой тип массива (с тем же самым числом измерений), если тип элемента первого массива более определен, чем тип элемента второго массива.

- Иначе, если один элемент является нерасширенной операцией, а другой — расширенной операцией, то нерасширенная операция является лучшим элементом.
- Иначе никакой функциональный элемент не является лучшим.

7.5.3.3. Лучшее приведение из выражения

Если задано неявное приведение C_1, которое преобразует из выражения E к типу T_1, и неявное приведение C_2, которое преобразует из выражения E к типу T_2, то C_1 является лучшим приведением, чем C_2, если выполняется по меньшей мере один из следующих пунктов:

- E имеет тип S и существует тождественное преобразование из S к T_1, но не из S к T_2.
- E не является анонимной функцией и является лучшим целевым объектом для приведения, чем T_2 (раздел 7.5.3.5).
- E является анонимной функцией, T_1 является либо типом делегата D_1, либо типом дерева выражений Expression<D_1>, T_2 является либо типом делегата D_2, либо типом дерева выражений Expression<D_2>, и выполняется один из следующих пунктов:
  - D_1 является лучшей целью приведения, чем D_2.
  - D_1 и D_2 имеют идентичные списки параметров и выполняется один из следующих пунктов:
    - D_1 имеет тип возвращаемого значения Y_1, а D_2 имеет тип возвращаемого значения Y_2, выведенный тип возвращаемого значения X существует в контексте данного списка параметров (раздел 7.5.2.12), и приведение от X к Y_1 лучше, чем приведение от X к Y_2.
    - D_1 имеет тип возвращаемого значения Y, а D_2 не возвращает никакого значения.

7.5.3.4. Лучшее приведение из типа

Если дано приведение C_1, которое преобразует из типа S к типу T_1, и приведение C_2, которое преобразует из типа S к типу T_2, C_1 является лучшим приведением, чем C_2, если по меньшей мере одно из следующих условий выполняется:

- Существует тождественное преобразование из типа S к типу T_1, но не от S к T_2.
- T_1 является лучшей целью приведения, чем T_2 (раздел 7.5.3.5).

7.5.3.5. Лучший целевой объект приведения

Заданы два различных типа T_1 и T_2, T_1 является лучшим целевым объектом приведения, чем T_2, если выполняется по меньшей мере одно из следующих условий:

- Существует неявное приведение из T_1 к T_2, и не существует неявного приведения из T_2 к T_1.
- T_1 является целым типом со знаком, а T_2 является целым типом без знака. Более подробно:
  - T_1 есть `sbyte`, а T_2 есть `byte`, `ushort`, `uint` или `ulong`.
  - T_1 есть `short`, а T_2 есть `ushort`, `uint` или `ulong`.
  - T_1 есть `int`, а T_2 есть `ulong`.

7.5.3.6. Перегрузка в обобщенных классах

Хотя объявленные сигнатуры должны быть уникальными, возможно, что подстановка аргументов типа может в результате дать идентичные сигнатуры. В таких случаях будет выбран наиболее определенный элемент в соответствии с правилами разрушения связи для разрешения перегрузки, приведенным ранее.

Следующий пример показывает образцы допустимой и недопустипой перегрузки в соответствии с этим правилом:

```
interface I1<T> {...}
interface I2<T> {...}
class G1<U>
{
   int F1(U u);                // Разрешение перегрузки для G<int>.F1
   int F1(int i);              // будет выбран необобщенный тип

   void F2(I1<U> a);           // Допустимая перегрузка
   void F2(I2<U> a);
}
class G2<U,V>
{
   void F3(U u, V v);          // Допустимо, но разрешение перегрузки для
   void F3(V v, U u);          // G2<int,int>.F3 не будет выполнено

   void F4(U u, I1<V> v);      // Допустимо, но разрешение перегрузки для
   void F4(I1<V> v, U u);      // G2<I1<int>,int>.F4 не будет выполнено

   void F5(U u1, I1<V> v2);    // Допустимая перегрузка
   void F5(V v1, U u2);

   void F6(ref U u);           // Допустимая перегрузка
   void F6(out V v);
}
```

7.5.4. Проверка динамического разрешения перегрузки во время компиляции

Для большинства динамически связанных операций набор возможных кандидатов для разрешения неизвестен во время компиляции. В некоторых случаях, однако, он известен во время компиляции. Перечислим эти случаи:

- Вызовы статического метода с динамическими аргументами.
- Вызовы метода экземпляра, когда получатель не является динамическим выражением.
- Вызовы конструктора с динамическими аргументами.

В этих случаях во время компиляции выполняется ограниченная проверка каждого кандидата, чтобы понять, может ли он оказаться подходящим во время выполнения программы. Эта проверка включает в себя следующие шаги:

- Частичное выведение типов: любой аргумент типа, не зависящий прямо или косвенно от аргумента типа `dynamic`, выводится согласно правилам раздела 7.5.2. Остальные аргументы типа являются *неизвестными*.
- Частичная проверка на пригодность: пригодность кандидатов проверяется в соответствии с разделом 7.5.3.1, но параметры, типы которых *неизвестны*, игнорируются.

Если никакой кандидат не проходит этот тест, выдается ошибка компиляции.

7.5.5. Вызов функционального элемента

Этот раздел описывает процесс, который происходит во время выполнения программы при вызове функционального элемента. Предполагается, что во время связывания элемент, который должен быть вызван, однозначно определен, возможно, применением разрешения перегрузки к набору возможных функциональных элементов.

Для удобства описания процесса вызова функциональные элементы подразделяют на две категории:

- Статические функциональные элементы. Это конструкторы экземпляров, статические методы, коды доступа статических свойств и операции, определенные пользователем. Статические функциональные элементы всегда невиртуальные.
- Функциональные элементы экземпляров. Это методы экземпляров, коды доступа свойств экземпляра и коды доступа индексаторов. Функциональные элементы экземпляров могут быть как невиртуальными, так и виртуальными, и они всегда вызываются для отдельного экземпляра. Экземпляр вычисляется при помощи выражения экземпляра, и он становится доступным внутри функционального элемента как `this` (раздел 7.6.7).

Обработка вызова функционального элемента во время выполнения программы состоит из нескольких шагов, приведенных далее. Пусть `M` — функциональный элемент, и если `M` является элементом экземпляра, то `E` — выражение экземпляра:

- Если `M` является статическим функциональным элементом:
  - Вычисляется список аргументов, как описано в разделе 7.5.1.
  - Вызывается `M`.
- Если `M` есть функциональный элемент экземпляра, объявленный в *типе-значении*:
  - Вычисляется `E`. Если вычисление приводит к исключению, дальнейшие шаги не выполняются.

- Если `E` не классифицирована как переменная, то создается временная логическая переменная типа `E`, и значение `E` присваивается этой переменной. `E` затем переклассифицируется как ссылка на эту временную локальную переменную. Временная переменная доступна как `this` внутри `M`, но не доступна любым другим способом. Таким образом, только когда `E` является подлинной переменной, вызывающая сторона может наблюдать изменения, которые `M` производит с `this`.

> **ЭРИК ЛИППЕРТ**
>
> Этот пункт иллюстрирует еще один способ, при использовании которого сочетание изменчивости (mutability) и значимой семантики копирования может привести к неприятностям. Например, поле `readonly` не классифицировано как переменная после выполнения конструктора. Таким образом, попытка вызвать метод, который изменяет содержимое поля `readonly` типа `value`, будет успешной, но в результате действительные изменения произойдут с копией! Этой проблемы можно избежать, если вообще стараться не использовать изменяемые типы `value`.

- Вычисляется список аргументов, как описано в разделе 7.5.1.
- Вызывается `M`. Переменная, на которую ссылается `E`, становится переменной, на которую ссылается `this`.

- Если `M` есть функциональный элемент экземпляра, объявленный в *ссылочном-типе*:
  - Вычисляется `E`. Если вычисление приводит к исключению, дальнейшие шаги не выполняются.
  - Вычисляется список аргументов, как описано в разделе 7.5.1.
  - Если тип `E` является *типом-значением*, выполняется преобразование упаковки (раздел 4.3.1), чтобы привести `E` к типу `object`, и считается, что `E` имеет тип `object` во время следующих шагов. В этом случае `M` может быть только элементом `System.Object`.
  - Значение `E` проверяется на допустимость. Если значением `E` является `null`, выдается исключение `NullReferenceException` и дальнейшие шаги не выполняются.
  - Реализация функционального элемента, которая будет вызвана, определяется следующими условиями:
    - Если определенный во время связывания тип `E` есть интерфейс, функциональный элемент, который будет вызван, есть реализация `M`, обеспеченная типом экземпляра, на который ссылается `E`, определенным во время выполнения программы.
    - Иначе, если `M` является виртуальным функциональным элементом, функциональный элемент, который должен быть вызван, есть реализация `M`, обеспеченная типом экземпляра, на который ссылается `E`. Этот функциональный элемент определяется с помощью правил соответствия

интерфейса (раздел 10.6.3), определяющих реализацию `M`, которая задана в типе времени выполнения экземпляра, на который ссылается `E`.

- Иначе `M` является невиртуальным функциональным элементом, и он и является тем элементом, который будет вызван.

- Вызывается реализация функционального элемента, определенная на предыдущем шаге. Объект, на который ссылается `E`, становится объектом, на который ссылается `this`.

7.5.5.1. Вызовы для упакованных экземпляров

Функциональный элемент, реализованный в *типе-значении*, может быть вызван через упакованный экземпляр этого типа в следующих ситуациях:

- Когда функциональный элемент является переопределением (`override`) метода, унаследованного от типа `object`, и вызывается с помощью выражения экземпляра типа `object`.
- Когда функциональный элемент является реализацией интерфейсного функционального элемента и вызывается с помощью выражения экземпляра *интерфейсного-типа*.
- Когда функциональный элемент вызывается с помощью делегата.

В этих ситуациях упакованный экземпляр считается содержащим переменную *типа-значения*, и на эту переменную ссылается `this` внутри вызова функционального элемента. В частности, когда функциональный элемент вызывается для упакованного экземпляра, он может модифицировать значение, содержащееся в этом упакованном экземпляре.

7.6. Первичные выражения

Первичные выражения содержат простейшие формы выражений.

```
первичное-выражение:
   первичное-выражение-без-создания массива
   выражение-создания-массива

первичное-выражение-без-создания-массива:
   литерал
   простое-имя
   выражение-в-скобках
   доступ-к-элементу
   выражение-вызова
   доступ-к-элементу-массива
   this-доступ
   base-доступ
   пост-инкрементное-выражение
   пост-декрементное-выражение
   выражение-создания-объекта
   выражение-создания-делегата
```

```
выражение-создания-анонимного-объекта
выражение-typeof
checked-выражение
unchecked-выражение
выражение-значения-по-умолчанию
выражение-анонимного-метода
```

Первичные выражения подразделяются на *выражения-создания-массива* и *первичные-выражения-без-создания-массива*. Рассмотрение выражения создания массива таким образом вместо представления его наряду с другими простыми формами выражений дает возможность в грамматике не допускать подверженного ошибкам кода, подобного

```
object o = new int[3][1];
```

который иначе мог бы интерпретироваться как

```
object o = (new int[3])[1];
```

7.6.1. Литералы

Первичное-выражение, состоящее из *литерала* (раздел 2.4.4), классифицируется как значение.

7.6.2. Простые имена

Простое-имя состоит из идентификатора, за которым следует не обязательный список аргументов-типов:

простое-имя:
  идентификатор *список-аргументов-типов*$_{opt}$

Простое-имя может быть либо в форме `I`, либо в форме `I<A`$_1$`,...,A`$_K$`>`, где `I` есть одиночный идентификатор, а `<A`$_1$`,...,A`$_K$`>` есть необязательный *список аргументов-типов*. Если список аргументов-типов не определен, `K` считается равным нулю. *Простое имя* вычисляется и классифицируется следующим образом:

- Если `K` равно нулю и *простое-имя* появляется внутри *блока* и область объявлений локальных переменных *блока* (или объемлющего *блока*) (раздел 3.3) содержит локальную переменную, параметр или константу с именем `I`, то это *простое-имя* ссылается на эту локальную переменную, параметр или константу и классифицируется как переменная или значение.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Это правило также применимо, если простое имя появляется внутри *инициализатора конструктора* и соответствует имени параметра этого конструктора.

- Если `K` равно нулю и *простое-имя* появляется внутри тела объявления обобщенного метода и объявление содержит параметр-тип с именем `I`, то *простое-имя* ссылается на этот параметр-тип.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Это условие всегда впоследствии приводит к ошибке компиляции.

- Иначе, для каждого типа экземпляра `T` (раздел 10.3.1), начиная с типа экземпляра, объявление которого непосредственно содержит данный экземпляр, и далее переходя к типам экземпляров каждого из объявлений объемлющих классов или структур (если таковые имеются), выполняются следующие условия:
  - Если `K` равно нулю и объявление `T` содержит параметр-тип с именем `I`, то *простое-имя* ссылается на этот параметр-тип.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Это условие всегда впоследствии приводит к ошибке компиляции.

  - Иначе, если поиск элемента (раздел 7.4) с простым именем `I` в `T` с `K` аргументов типа приводит к соответствию:
    - Если `T` является типом экземпляра класса или структуры, непосредственно содержащего (содержащей) данный экземпляр, и поиск находит один метод или более, результатом будет группа методов с ассоциированным выражением экземпляра `this`. Если список аргументов был определен, он используется при вызове обобщенного метода (раздел 7.6.5.1).
    - Иначе, если `T` является типом экземпляра класса или структуры, непосредственно содержащего (содержащей) данный экземпляр, если поиск находит элемент экземпляра и если ссылка встречается внутри *блока* конструктора экземпляра, метода экземпляра или кода доступа экземпляра, результат будет таким же, как в случае доступа к элементу (раздел 7.6.4) в форме `this.I`. Такое может случиться, только когда `K` равно нулю.
    - Иначе результат будет таким же, как в случае доступа к элементу (раздел 7.6.4) в форме `T.I` или `T.I<`A_1`, ..., `A_K`>`. При этом, если *простое-имя* ссылается на элемент экземпляра, выдается ошибка времени связывания.
- Иначе для каждого пространства имен `N`, начиная с пространства имен, в котором находится *простое-имя*, далее переходя к каждому из объемлющих пространств имен (если таковые существуют) и заканчивая глобальным пространством имен, выполняются следующие шаги до тех пор, пока сущность не будет определена:
  - Если `K` равно нулю и `I` есть имя пространства имен в `N`, то:
    - Если *простое-имя* находится в объявлении пространства имен для `N` и объявление пространства имен содержит *директиву-внешнего-псевдонима* или *using-директиву-псевдонима*, которая связывает имя I

с пространством или типом, то *простое-имя* неоднозначно и выдается ошибка компиляции.

▪ Иначе простое имя ссылается на пространство имен, именуемое `I`, в `N`.

- Иначе, если `N` содержит достижимый тип, имеющий имя `I` и `K` параметров типа, то:

▪ Если `K` равно нулю и *простое-имя* находится в объявлении пространства имен для `N` и объявление пространства имен содержит *директиву-внешнего-псевдонима* или *using-директиву-псевдонима*, которая связывает имя `I` с пространством или типом, то *простое имя* неоднозначно и выдается ошибка компиляции.

▪ Иначе *имя-пространства-имен-или-типа* ссылается на тип, сконструированный с данными аргументами-типами.

- Иначе, если местоположение простого имени находится в объявлении пространства имен для `N`:

▪ Если `K` равно нулю и объявление пространства имен содержит *директиву-внешнего-псевдонима* или *директиву-using-для-псевдонима*, которая связывает имя `I` с импортированным пространством имен или типом, то это простое имя ссылается на это пространство имен или тип.

▪ Иначе, если пространства имен, импортированные в данное объявление пространства имен с помощью *using-директив-пространства-имен,* содержат в точности один тип, имеющий имя `I` и `K` параметров-типов, то *простое-имя* ссылается на этот тип, сконструированный с даннымми аргументами-типами.

▪ Иначе, если пространства имен, импортированные в данное объявление пространства имен с помощью *using-директив-пространства-имен* содержат более одного типа, имеющего имя `I` и `K` параметров-типов, то *простое-имя* неоднозначно и выдается ошибка.

Заметим, что этот шаг является полностью аналогичным соответствующему шагу в обработке *имени-пространства-имен-или-типа* (раздел 3.8).

- Иначе *простое-имя* не определено и выдается ошибка компиляции.

7.6.2.1. Инвариантное значение в блоках

Для любого вхождения данного идентификатора в качестве *простого-имени* в выражение или описателя внутри области объявлений локальной переменной (раздел 3.3), непосредственно содержащей это вхождение, любое другое вхождение того же самого идентификатора в качестве *простого-имени* в выражении или описателе должно ссылаться на тот же объект. Это правило дает уверенность в том, что значение имени всегда одно и то же внутри данного блока, блока `switch`, оператора `for`, оператора `foreach`, оператора `using` или анонимной функции.

> **ЭРИК ЛИППЕРТ**
> Одним из наиболее тонких конструктивных следствий этого правила является то, что стало более безопасным выполнять рефакторинг, который включает перемещение объявлений локальных переменных. Любой рефакторинг, который приведет к изменению семантики простого имени, будет «отловлен» компилятором.

Пример:

```
class Test
{
   double x;

   void F(bool b) {
      x = 1.0;
      if (b) {
         int x;
         x = 1;
      }
   }
}
```

в результате дает ошибку компиляции, поскольку `x` ссылается на различные сущности внутри внешнего блока (область которого содержит вложенный блок в операторе `if`). Напротив, пример:

```
class Test
{
   double x;

   void F(bool b) {
      if (b) {
         x = 1.0;
      }
      else {
         int x;
         x = 1;
      }
   }
}
```

является допустимым, поскольку имя `x` не используется во внешнем блоке.

Отметим, что это правило инвариантного значения применимо только к простым именам. Для одного и того же идентификатора вполне допустимо иметь одно значение в качестве простого имени и другое значение в качестве правого операнда доступа к элементу (раздел 7.6.4). Например:

```
struct Point
{
   int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Этот пример иллюстрирует образец использования имен полей в качестве имен параметров конструктора экземпляра. В этом примере простые имена `x` и `y` ссылаются на параметры, но это не препятствует выражениям доступа к элементу `this.x` и `this.y` выполнять доступ к полям.

7.6.3. Выражения в скобках

Выражение-в-скобках состоит из выражения, заключенного в круглые скобки:

```
выражение-в-скобках:
   ( выражение )
```

Выражение-в-скобках вычисляется путем вычисления *выражения*, находящегося внутри скобок. Если выражение внутри скобок указывает на пространство имен или тип, выдается ошибка компиляции. В противном случае результатом вычисления *выражения-в-скобках* является результат вычисления заключенного в скобки *выражения*.

7.6.4. Доступ к элементу

Доступ-к-элементу (*member-access*) состоит из *первичного-выражения*, *предопределенного-типа* или *уточненного-псевдонима-элемента*, за которым следует лексема «.» с последующим идентификатором, за которым следует необязательный *список-аргументов-типов*.

```
доступ-к-элементу:
   первичное-выражение . идентификатор список-аргументов-типов opt
   предопределенный-тип . идентификатор список-аргументов-типов opt
   уточненный-псевдоним-элемента . идентификатор список-аргументов-типов opt

предопределенный-тип: один из
   bool     byte    char    decimal  double  float   int     long
   object   sbyte   short   string   uint    ulong   ushort
```

Правила для *уточненного-псевдонима-элемента* приведены в разделе 9.7.

Доступ-к-элементу может быть либо в форме `E.I`, либо в форме $E.I<A_1,...,A_K>$, где `E` является *первичным-выражением*, `I` является одиночным идентификатором, а $<A_1,...,A_K>$ является необязательным списком аргументов-типов.

Доступ-к-элементу с *первичным-выражением* типа `dynamic` является динамически связанным (раздел 7.2.2). В этом случае компилятор классифицирует доступ к элементу как доступ к свойству типа `dynamic`. Приведенные ниже правила, определяющие значение *доступа-к-элементу*, затем применяются во время выполнения программы, используя тип *первичного-выражения* времени выполнения вместо типа времени компиляции. Если такая классификация во время выполнения приводит к группе методов, то доступ к элементу должен быть *первичным-выражением выражения-вызова*.

Доступ-к-элементу вычисляется и классифицируется следующим образом:

- Если `K` равно нулю, `E` является пространством имен и `E` содержит вложенное пространство имен `I`, то результатом будет это пространство имен.

- Иначе, если `E` является пространством имен и `E` содержит достижимый тип, имеющий имя `I` и `K` параметров-типов, то результатом будет тип, сконструированный с данными аргументами-типами.
- Если `E` является *предопределенным-типом* или *первичным-выражением*, классифицированным как тип, и если `E` не является параметром-типом, и если поиск элемента (раздел 7.4) `I` в `E` с `K` параметрами-типами приводит к соответствию, то `E.I` классифицируется и вычисляется следующим образом:
  - Если `I` определяет тип, то результатом будет этот тип, сконструированный с данными аргументами-типами.
  - Если `I` определяет один метод или более, то результатом будет группа методов без связанного с ней выражения экземпляра. Если определен список аргументов-типов, он будет использоваться при вызове обощенного метода (раздел 7.6.5.1).
  - Если `I` определяет статическое свойство, то результатом будет доступ к свойству без связанного с ним выражения экземпляра.
  - Если `I` определяет статическое поле:
    - Если это поле является `readonly` и ссылка встречается вне статического конструктора класса или структуры, где описано это поле, то результатом будет значение, а именно значение статического поля `I` в `E`.
    - Иначе результатом будет переменная, а именно статическое поле `I` в `E`.
  - Если `I` определяет статическое событие:
    - Если ссылка появляется внутри класса или структуры, в которых объявлено событие и событие было объявлено без *объявлений кодов доступа* (раздел 10.8), то `E.I` обрабатывается в точности так же, как если бы `I` было статическим полем.
    - Иначе результатом будет доступ к событию без ассоциированного выражения экземпляра.
  - Если `I` определяет константу, то результатом будет значение, а именно значение этой константы.
  - Если `I` определяет элемент перечисления, то результатом будет значение, а именно значение этого элемента перечисления.
  - Иначе `E.I` является недопустимой ссылкой и выдается ошибка компиляции.
- Если `E` является доступом к свойству, доступом к индексатору, переменной или значением типа `T` и поиск элемента (раздел 7.4) `I` в `T` с `K` аргументами-типами находит соответствие, то `E.I` вычисляется и классифицируется следующим образом:
  - Во-первых, если `E` является доступом к свойству или индексатору, то значение доступа к свойству или индексатору будет получено (раздел 7.1.1), и `E` будет переклассифицировано как значение.
  - Если `I` определяет один метод или более, то результатом будет группа методов с ассоциированным выражением экземпляра `E`. Если список аргументов-

типов был определен, он используется при вызове обобщенного метода (раздел 7.6.5.1).

- Если `I` определяет свойство экземпляра, то результатом будет доступ к свойству с ассоциированным выражением экземпляра `E`.
- Если `T` является *классом* и `I` определяет поле экземпляра этого *класса*:
  - Если значение `E` есть `null`, то выбрасывается исключение `System.NullReferenceException`.
  - Иначе, если поле является `readonly` и ссылка существует вне конструктора экземпляра того класса, в котором поле было объявлено, то результатом будет значение, а именно значение поля `I` в объекте, на который ссылается `E`.
  - Иначе результатом будет переменная, а именно поле `I` в объекте, на который ссылается `E`.
- Если `T` является *структурой* и `I` идентифицирует поле экземпляра *структуры*:
  - Если `E` является значением или если поле является `readonly` и ссылка встречается вне конструктора экземпляра структуры, в которой объявлено поле, то результатом будет значение, а именно значение поля в экземпляре структуры, заданном `E`.
  - Иначе результатом будет переменная, а именно поле I в экземпляре структуры, заданном `E`.
- Если `I` определяет событие экземпляра :
  - Если ссылка встречается внутри класса или структуры, в котором (в которой) объявлено событие, и это событие было объявлено без *объявлений кодов доступа* (раздел 10.8), и ссылка появляется не как левая часть операции `+=` или `-=`, то `E.I` обрабатывается в точности так же, как если бы `I` было полем экземпляра.
  - Иначе результатом будет доступ к событию с ассоциированным выражением экземпляра `E`.

- Иначе выполняется попытка обработать `E.I` как вызов метода расширения (раздел 7.6.5.2). Если она заканчивается неудачно, `E.I` является недопустимой ссылкой и выдается ошибка времени связывания.

ПИТЕР СЕСТОФТ

Два пункта только что приведенного списка, утверждающие «Если поле является `readonly`... то результатом будет значение» могут преподнести небольшой сюрприз, когда поле имеет тип структуры, и эта структура имеет изменяемое поле (*не* рекомендуемая комбинация — см. другие аннотации к этому пункту). Рассмотрим следующий пример:

```
struct S {
   public int x;
   public void SetX() { x = 2; }
}
```

продолжение ⇗

```
class C {
   static S s;
   public static void M() { s.SetX(); }
}
```

Как ожидалось, после вызова `s.SetX()` поле структуры `s.x` имеет значение `2`. Теперь если мы добавим модификатор `readonly` к описанию поля `s`, то внезапно вызов `s.SetX()` перестает иметь действие! То есть теперь в вызове метода `s` является значением, а не переменной, согласно правилам, данным ранее; таким образом, `SetX()` выполняется над *копией* поля `s`, а не над самим полем `s`. Несколько удивительный результат получается, если вместо `s` будет локальная переменная структурного типа, объявленная в операторе `using` (раздел 8.13), которая также имеет свойство делать `s` изменяемой; тогда `s.SetX()` обновляет `s.x`, как ожидалось.

7.6.4.1. Идентичные простые имена и имена типов

При доступе к элементу в форме `E.I`, если `E` является одиночным идентификатором и если значением `E` в качестве *простого-имени* (раздел 7.6.2) является константа, поле, свойство, локальная переменная или параметр того же типа, что у значения `E` в качестве *имени-типа* (раздел 3.8), то оба возможных значения `E` допустимы. Два возможных значения `E.I` никогда не являются неоднозначными, поскольку `I` в обоих случаях должно быть элементом типа `E`. Иными словами, это правило просто разрешает доступ к статическим элементам и вложенным типам `E`, там, где иначе выдавалась бы ошибка компиляции. Например:

```
struct Color
{
   public static readonly Color White = new Color(...);
   public static readonly Color Black = new Color(...);

   public Color Complement() {...}
}
class A
{
   public Color Color;            // Поле Color типа Color
   void F() {
      Color = Color.Black;    // Ссылается на статический элемент Color.Black
      Color = Color.Complement(); // Вызывает Complement() для поля Color
   }
   static void G() {
      Color c = Color.White;   // Ссылается на статический элемент Color.White
   }
}
```

Идентификаторы, которые ссылаются на тип `Color` внутри класса `A`, выделены жирным, а те, которые ссылаются на поле `Color`, не выделены.

7.6.4.2. Грамматическая неоднозначность

Грамматические правила для *простых-имен* (раздел 7.6.2) и *доступа-к-элементу* (раздел 7.6.4) могут приводить к неоднозначности в грамматике для выражений.

Например, оператор

```
F(G<A,B>(7));
```

может быть интерпретирован как вызов `F` с двумя аргументами, `G < A and B > (7)`. В то же время он может быть интерпретирован как вызов `F` с одним аргументом, который является вызовом обобщенного метода `G` с двумя аргументами-типами и с одним обычным аргументом.

Если последовательность лексем может быть разобрана (в контексте) как *простое-имя* (раздел 7.6.2), *доступ-к-элементу* (раздел 7.6.4) или *доступ-к-элементу-через-указатель* (раздел 18.5.2), заканчиваясь *списком-аргументов-типов* (раздел 4.4.1), проверяется лексема, непосредственно следующая за закрывающим символом `>`. Если она является одной из

```
(    )    ]    }    :    ;    ,    .    ?    ==    !=    |    ^
```

то список аргументов-типов сохраняется как часть *простого-имени*, *доступа-к-элементу* или *доступа-к-элементу-через-указатель*, и любые другие возможные толкования последовательности лексем отвергаются. В противном случае список аргументов-типов не рассматривается как часть *простого-имени*, *доступа-к-элементу* или *доступа-к-элементу-через-указатель*, даже если нет других возможных толкований последовательности лексем. Отметим, что эти правила неприменимы для разбора *списка-аргументов-типа* в *имени-пространства-имен-или типа* (раздел 3.8). Каждый из операторов

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

может быть интерпретирован как вызов `F` с двумя аргументами.

Оператор

```
x = F < A > +y;
```

будет интерпретирован как включающий в себя операцию «меньше чем», операцию «больше чем» и операцию «унарный плюс», как если бы оператор был записан в виде `x = (F < A) > (+y)`, вместо того чтобы *простое-имя* со *списком-аргументов-типа* следовало за бинарной операцией плюс. В операторе

```
x = y is C<T> + z;
```

лексемы `C<T>` интерпретируются как *имя-пространства-имен-или-типа* со *списком-аргументов-типов*.

7.6.5. Выражения вызова

Выражения-вызова используются для вызова метода.

```
выражение-вызова:
   первичное-выражение   (   список-аргументов_opt   )
```

Выражение-вызова является динамически связанным (раздел 7.2.2), если выполняется по меньшей мере один из следующих пунктов:

- *Первичное-выражение* имеет тип времени компиляции `dynamic`.
- По меньшей мере один аргумент необязательного *списка-аргументов* имеет тип времени компиляции `dynamic`, и *первичное-выражение* не является делегатом.

ВЛАДИМИР РЕШЕТНИКОВ

Одно исключение из этого правила: аргументы `ref/out` типа `dynamic` не приводят к динамическому связыванию.
Если *выражение-вызова* динамически связано и его *первичное-выражение* обозначает группу методов, которая получена в результате *base-доступа*, то выдается ошибка компиляции (CS1971).

В этом случае компилятор классифицирует *выражение-вызова* как значение типа `dynamic`. Приведенные далее правила определения значения *выражения-вызова* применяются во время выполнения программы, используя тип времени выполнения для *первичного-выражения* и аргументов, имеющих тип времени компиляции `dynamic`. Если *первичное-выражение* не имело тип времени компиляции `dynamic`, то для вызова метода во время компиляции выполняется частичная проверка, как описано в разделе 7.5.4.

Первичное-выражение для *выражения-вызова* должно быть группой методов или значением *типа-делегата*. Если *первичное-выражение* является группой методов, *выражение-вызова* является вызовом метода (раздел 7.6.5.1). Если *первичное-выражение* является значением *типа-делегата*, выражение вызова является вызовом делегата (раздел 7.6.5.3). Если *первичное-выражение* не является ни группой методов, ни значением *типа-делегата*, выдается ошибка времени связывания.

Необязательный *список-аргументов* (раздел 7.5.1) задает значения или ссылки на переменные для параметров метода.

Результат вычисления *выражения-вызова* классифицируется следующим образом:

- Если *выражение-вызова* вызывает метод или делегат, который возвращает `void`, результатом будет пустое выражение. Выражение, которое классифицируется как пустое выражение, разрешено только в контексте *оператора выражения* (раздел 8.6) или как тело *лямбда-выражения* (раздел 7.15). В противном случае выдается ошибка компиляции.
- Иначе результатом будет значение типа, возвращаемого методом или делегатом.

7.6.5.1. Вызов метода

Для вызова метода *первичное-выражение* для *выражения-вызова* должно быть группой методов. Группа методов определяет один метод, который должен быть вызван, или набор перегруженных методов, из которых выбирается определенный метод, который должен быть вызван. В этом последнем случае определение этого отдельного метода, который будет вызван, основывается на контексте, обеспеченном типами аргументов в *списке-аргументов*.

Обработка вызова метода в форме `M(A)` во время связывания, где `M` есть группа методов (возможно, содержащая список аргументов-типов), а `A` есть необязательный список аргументов, состоит из следующих шагов:

- Создается набор кандидатов для вызова метода. Для каждого метода `F`, ассоциированного с группой методов `M`:

- Если F не обощенный, F является кандидатом, когда:
  - M не имеет аргументов-типов, и
  - F применим в соответствии с A (раздел 7.5.3.1).
- Если F обобщенный, а M не имеет списка аргументов, F является кандидатом, когда:
  - выведение типа (раздел 7.5.2) проходит успешно, в результате получается список аргументов-типов для вызова, и
  - найденные аргументы-типы замещают параметры-типы соответствующего метода, все сконструированные типы в списке параметров F удовлетворяют их ограничениям (раздел 4.4.4), и список параметров F применим в соответствии с A (раздел 7.5.3.1).

ВЛАДИМИР РЕШЕТНИКОВ

Заметьте, что на этом шаге проверяются только ограничения сконструированных типов, заданных в списке параметров. Ограничения, относящиеся непосредственно к обобщенному методу, проверяются позже, во время завершающей фазы.

- Если F обобщенный, а M имеет список аргументов, F является кандидатом, когда:
  - F имеет такое же количество параметров-типов, как в списке аргументов-типов, и
  - найденные аргументы-типы замещают параметры-типы соответствующего метода, все сконструированные типы в списке параметров F удовлетворяют их ограничениям (раздел 4.4.4), и список параметров F применим в соответствии с A (раздел 7.5.3.1).

- Набор возможных методов сокращается до методов из наиболее производных типов: для каждого метода C.F из набора, где C — тип, в котором объявлен метод F, все методы, объявленные в базовом типе, удаляются из набора. Кроме того, если C является классом, отличным от object, все методы, объявленные в интерфейсном типе, удаляются из набора. (Это последнее правило действует, только когда группа методов стала результатом поиска элемента в параметре-типе, имеющем эффективный базовый класс, отличный от object, и непустой набор эффективных интерфейсов.)

ВЛАДИМИР РЕШЕТНИКОВ

Это правило означает, что выбирается подходящий метод из наиболее производного класса, даже если в родительском типе существует метод с более подходящими типами параметров и даже если выбранный из наиболее производного типа метод не пройдет окончательную проверку. Также помните, что все переопределения были удалены до этого шага, во время поиска элемента (раздел 7.4).

```
class Base
{
   public virtual void Foo(int x) { }
}
```

продолжение ⇨

```
class Derived : Base
{
   public override void Foo(int x) { }

   static void Foo(object x) { }

   static void Main()
   {
      var d = new Derived();
      d.Foo(1);

      // Error CS0176: Member ‘Derived.Foo(object)’
      // cannot be accessed with an instance reference;
      // qualify it with a type name instead
      // Ошибка CS0176: Элемент ‘Derived.Foo(object)’
      // не может быть доступен через ссылку на экземпляр;
      // для уточнения имени задайте тип
   }
}
```

- Если полученный набор возможных методов пустой, дальнейшая обработка на следующих шагах не выполняется, и вместо этого предпринимается попытка обработать вызов как вызов метода расширения (раздел 7.6.5.2). Если она завершается неудачно, то подходящих методов не существует и выдается ошибка времени связывания.
- Наилучший метод из набора возможных методов находится путем применения правил разрешения перегрузки раздела 7.5.3. Если единственный лучший метод нельзя найти, значит, вызов метода неоднозначен и выдается ошибка времени связывания. Когда выполняется разрешение перегрузки, параметры обобщенного метода рассматриваются после подстановки аргументов-типов (заданных или выведенных) для соответствующих типов параметров метода.
- Выполняется окончательная проверка выбранного наилучшего метода:
  - Метод проверяется в контексте группы методов: если наилучший метод является статическим методом, группа методов должна получаться в результате из *простого-имени* или *доступа-к-элементу* через тип. Если наилучший метод является методом экземпляра, группа методов должна получаться в результате из *простого-имени* или *доступа-к-элементу* через переменную, значение или *base-доступ*. Если никакое из этих требований не выполняется, выдается ошибка времени связывания.

ВЛАДИМИР РЕШЕТНИКОВ

Если наилучший метод является методом экземпляра и группа методов получается из *простого-имени*, то *простое-имя* не должно появляться в статическом контексте (то есть в статических элементах, вложенных типах, полях экземпляра, в подобных полям

инициализаторах событий или в *инициализаторах-конструкторов*), иначе будет выдана ошибка времени связывания.

- Если наилучший метод является обобщенным методом, аргументы-типы (заданные или выведенные) проверяются на соблюдение ограничений, объявленных в обобщенном методе. Если хотя бы один аргумент-тип не удовлетворяет соответствующему ограничению (ограничениям) параметра-типа, выдается ошибка времени связывания.

ВЛАДИМИР РЕШЕТНИКОВ

Если проверка проходит успешно, это автоматически гарантирует, что все ограничения на все сконструированные типы в типе возвращаемого значения и в теле метода также выполняются.

ЭРИК ЛИППЕРТ

Правило, описанное выше, породило множество споров: почему компилятор должен выполнить всю работу по выведению типов, разрешению перегрузки, удалению неподходящих кандидатов только для того, чтобы сказать: «Я выбрал метод, который после всего этого не работает»? Почему просто не считать, что методы, которые не удовлетворяют ограничениям, не являются даже кандидатами? Причина неочевидная, но важная: фундаментальный принцип построения C# состоит в том, что язык не должен предугадывать поведение пользователя. Если наилучший возможный выбор, следующий из аргументов и типов параметров, является недопустимым, то либо пользователь намеревался сделать наилучший выбор, но допустил некую ошибку, либо пользователь рассчитывал, что компилятор выберет что-нибудь другое. Наиболее безопасная вещь, которую можно сделать, — это предположить первое, чтобы не допустить неверных предположений.

Для выбранного и проверенного во время связывания на предыдущих шагах метода фактический вызов во время выполнения программы обрабатывается в соответствии с правилами вызова функционального элемента, описанными в разделе 7.5.4.

БИЛЛ ВАГНЕР

Следующий абзац является ключевым: методы расширения не могут изменять поведение, определенное создателем типа.

Интуитивно понятно, что действие описанных выше правил разрешения является следующим: чтобы найти метод, вызываемый данным вызовом метода, надо начать с типа, указанного в вызове метода, и продвигаться вверх по иерархии наследования до тех пор, пока не найдется по меньшей мере одно объявление подходящего, достижимого неперегруженного метода. Затем выполнить выведение типов и разрешение перегрузки на наборе подходящих, достижимых, неперегру-

женных методов, объявленных в этом типе, и вызов метода таким образом будет выбран. Если никакой метод не будет найден, попытаться обработать вызов как вызов метода расширения.

> **ЭРИК ЛИППЕРТ**
>
> Язык построен таким образом, что даже если «лучший» метод может быть найден в родительском классе, любой подходящий метод производного класса имеет приоритет. Есть две причины для такого решения.
>
> Во-первых, реализация производного класса предположительно имеет больше информации относительно заданной семантики этой операции в своем классе, чем реализация родительского класса; производный класс существует, поскольку он добавил функциональные возможности или специализацию к родительскому классу.
>
> Во-вторых, такая структура позволяет избежать одну из проблем семейства «хрупкого родительского класса». Предположим, у вас есть код, который рассчитан на вызов метода в производном классе. Если разрешение перегрузки выберет этот метод сегодня, то изменения в родительском классе завтра не приведут к тому, что во время следующей компиляции будет молча и внезапно выбран метод родительского класса.
>
> Конечно, эта схема также вводит противоположный сценарий: если вы зависите от разрешения перегрузки, для того чтобы выбрать метод базового класса, то некто, кто добавит метод к более производному классу, может точно так же незаметно изменить выбор разрешения перегрузки при перекомпиляции. Однако эта проблема «хрупкого производного класса» встречается в реальном действующем коде реже; по большей части желателен выбор более производного метода по причинам, изложенным выше.

> **ДЖОН СКИТ**
>
> Хотя я в основном согласен с аргументацией Эрика, такая структура не препятствует появлению неудачных ситуаций. Если производный класс вводит новую перегрузку метода родительского класса и *переопределяет метод родительского класса*, то ясно, что производный класс осведомлен о методе родительского класса — и несмотря на это, переопределенный вариант все так же не рассматривается при выборе метода, до тех пор пока алгоритм не достигнет родительского класса. Например, рассмотрим следующие методы в производном классе, если в базовом классе объявлен виртуальный метод:
>
> ```
> public void Foo(object x) { ... }
> public override void Foo(int y) { ... }
> ```
>
> Вызов Foo(10) для выражения производного типа будет выбирать первый метод — в противоположность ожиданиям любого разработчика, не имеющего отношения к Microsoft, которому я когда-либо представлял эту структуру.

7.6.5.2. Вызов метода расширения

При вызове метода (раздел 7.5.5.1) в одной из форм

выражение **.** *идентификатор* **( )**
выражение **.** *идентификатор* **(** *аргументы* **)**

```
выражение . идентификатор < аргументы-типы > ( )
выражение . идентификатор < аргументы-типы > ( аргументы )
```

если нормальная обработка вызова не находит подходящих методов, делается попытка обработать конструкцию как вызов метода расширения. Если *выражение* или любой из *аргументов* имеют тип времени компиляции `dynamic`, методы расширения не применяются.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Последнее правило неприменимо к аргументам `ref/out` типа `dynamic`.

Цель — найти наилучшее *имя-типа* `C`, такое, что произойдет соответствующий вызов статического метода:

```
C . идентификатор ( выражение )
C . идентификатор ( выражение , аргументы )
C . идентификатор < аргументы-типы > ( выражение )
C . идентификатор < аргументы-типы > ( выражение , аргументы )
```

Метод расширения $C_i.M_j$ может быть выбран, если:

- C_i является не обобщенным и не вложенным классом.
- Имя M_j является *идентификатором*.
- M_j является достижимым и подходящим, когда применяется к аргументам как статический метод, как описано выше.
- Существует неявное тождественное или ссылочное преобразование или преобразование упаковки из *выражения* к типу первого параметра M_j.

> **ЭРИК ЛИППЕРТ**
>
> Это правило обеспечивает то, что метод, который расширяет `double`, не будет расширять `int`. Это также гарантирует, что никакие методы расширения не будут определены для анонимных функций или групп методов.

Поиск для `C` выполняется следующим образом:

- Начиная с объявления непосредственно содержащего пространства имен, переходя затем к объявлениям каждого объемлющего пространства имен и заканчивая единицей компиляции, предпринимаются последовательные попытки для нахождения набора возможных методов расширения:
  - Если данное пространство имен или единица компиляции непосредственно содержит объявления необобщенных типов C_i с методами расширения M_j, которые могут быть выбраны, то набор этих методов расширения и есть набор кандидатов.
  - Если пространства имен, импортированные в данное пространство имен или единицу компиляции при помощи using-директив пространства имен, непосредственно содержат объявления необобщенных типов C_i с методами

расширения M_j, которые могут быть избраны, то набор этих методов расширения и есть набор кандидатов.

- Если набор кандидатов ни в одном объявлении пространства имен или единицы компиляции не найден, выдается ошибка компиляции.
- В противном случае к набору кандидатов применяется разрешение перегрузки, как описано в разделе 7.5.3. Если никакой единственный наилучший метод не найден, выдается ошибка компиляции.
- `C` является типом, внутри которого наилучший метод объявлен как метод расширения.

Используя `C` как целевой объект, вызов метода затем обрабатывается как вызов статического метода (раздел 7.5.4).

БИЛЛ ВАГНЕР

Эта технология несколько сложна, что усиливает желание придерживаться рекомендации не создавать копии методов расширения в различных пространствах имен.

ПИТЕР СЕСТОФТ

Заманчиво считать метод расширения этакой забавной разновидностью невиртуального метода экземпляра, которая вызывается только тогда, когда никакой обычный подходящий метод не доступен. К несчастью, метод расширения `SetX(this S s)` для структуры `S` не вполне то же самое, что метод экземпляра для `S`, поскольку вызов `s.SetX()` будет преобразован к вызову `SetX(s)`, который передает структуру `s` по значению, следовательно, копируя ее. Любой побочный эффект будет действовать на копию, а не на исходную структуру `s` — еще одна причина никогда не иметь изменяемых полей в структурных типах.

ДЖОН СКИТ

Один из вопросов относительно методов расширения — насколько легко случайно импортировать больше, чем вам нужно. Я бы предпочел сделать этот процесс более очевидным, например, с новым типом директивы `using`:

```
using static System.Linq.Enumerable;
```

Это позволяет библиотекам классов показывать методы расширения без добавления набора интеллектуальных средств IntelliSense, который может привести в смущение пользователя, вовсе не желавшего его применять. Кроме того, этот способ обеспечивает пояснения для сопровождающего программиста, предупреждающие, какие методы расширения можно ожидать увидеть внутри данной единицы компиляции.

Приведенные выше правила означают, что методы экземпляра имеют преимущество перед методами расширения, что методы расширения, достижимые в объявлениях внутренних пространств имен, имеют преимущество перед методами расширения, достижимыми в объявлениях внешних пространств имен, и что методы расширения, объявленные непосредственно в пространстве имен, имеют

преимущество перед методами расширения, импортированными в это же пространство с помощью using-директивы пространства имен. Например:

```
public static class E
{
      public static void F(this object obj, int i) { }
        public static void F(this object obj, string s) { }
}
class A { }
class B
{
   public void F(int i) { }
}
class C
{
   public void F(object obj) { }
}
class X
{
   static void Test(A a, B b, C c) {
      a.F(1);             // E.F(object, int)
      a.F("hello");       // E.F(object, string)

      b.F(1);             // B.F(int)
      b.F("hello");       // E.F(object, string)

      c.F(1);             // C.F(object)
      c.F("hello");       // C.F(object)
   }
}
```

В данном примере метод класса `B` имеет преимущество перед первым методом расширения, а метод класса `C` имеет преимущество перед обоими методами расширения.

```
public static class C
{
   public static void F(this int i) {
         Console.WriteLine("C.F({0})", i);
      }

   public static void G(this int i) {
         Console.WriteLine("C.G({0})", i);
      }

   public static void H(this int i) {
   Console.WriteLine("C.H({0})", i);
      }
   }

namespace N1
{
   public static class D
   {
      public static void F(this int i) {
      Console.WriteLine("D.F({0})", i);
   }
```

продолжение ⇗

```
      public static void G(this int i) {
      Console.WriteLine("D.G({0})", i);
      }
   }
}
namespace N2
{
   using N1;

   public static class E
   {
      public static void F(this int i) {
      Console.WriteLine("E.F({0})", i);
   }
}
   class Test
   {
      static void Main(string[] args)
      {
         1.F();
         2.G();
         3.H();
      }
   }
}
```

Выводом данного примера является:

```
E.F(1)
D.G(2)
C.H(3)
```

`D.G` имеет преимущество перед `C.G`, и `E.F` имеет преимущество перед `D.F` и `C.F`.

> **ДЖОЗЕФ АЛЬБАХАРИ**
>
> Одна из причин, по которой методы экземпляра имеют больший приоритет, чем методы расширения, — это желание иметь уверенность в том, что введение методов расширения в программу или платформу не нарушит существующий код.

> **ЭРИК ЛИППЕРТ**
>
> Способ, при помощи которого C# ищет методы расширения, приводит к новым проявлениям проблемы «хрупкого производного класса», упоминавшейся ранее. Если вы полагаетесь на обработку вызова, чтобы выбрать метод расширения, но некто вводит метод экземпляра с тем же именем в этом классе, то после перекомпиляции метод расширения уже не будет вызываться.
>
> Такое поведение на самом деле не является проблемой — обычно это в точности то, чего вы хотите. Метод класса имеет намного больше информации о внутренней структуре класса, чем статический метод расширения, так что у первого должно быть преимущество.

> Компилятор Microsoft C# не выдает предупреждения о том, что существует метод расширения, который мог бы быть альтернативой методу экземпляра. Если бы компилятор выдавал такое предупреждение, не было бы простого способа «исправить» это, кроме как разместить вокруг предупреждения директивы `#pragma`, чтобы его убрать.

> **ДЖОН СКИТ**
>
> Я думаю, в этом случае предупреждение действительно было бы наиболее подходящим действием, которое предупреждало бы разработчиков о возможности вызова метода с тем же именем, но с другим значением без изменения клиентского кода (после перекомпиляции). Во многих случаях, когда есть возможность перекомпилировать весь код, использующий методы расширения, и поэтому двоичная совместимость не требуется, — метод можно было бы полностью удалить или переименовать. В других случаях его можно преобразовать в нерасширяющий метод, что позволяет сохранить двоичную совместимость со старым кодом. Полностью игнорировать это изменение результата поиска элемента мне кажется неправильным.

7.6.5.3. Вызов делегата

Для вызова делегата *первичное-выражение* для *выражения-вызова* должно иметь значение *типа-делегата*. Кроме того, если рассматривать *тип-делегата* в качестве функционального элемента с тем же списком параметров, как у *типа-делегата*, *тип-делегата* должен быть применимым (раздел 7.5.3.1) для *списка-аргументов выражения-вызова*.

Обработка вызова делегата в форме `D(A)` во время выполнения программы, где `D` является первичным выражением типа делегата и `A` есть необязательный список параметров, состоит из следующих шагов:

- Вычисляется `D`. Если это вычисление приводит к исключению, дальнейшие шаги не выполняются.
- Значение `D` проверяется на допустимость. Если значение `D` есть `null`, генерируется `System.NullReferenceException` и дальнейшие шаги не выполняются.
- В противном случае `D` является ссылкой на экземпляр делегата. Вызовы функционального элемента (раздел 7.5.4) выполняются для каждой из вызываемых сущностей из списка вызова делегата. Для вызываемых сущностей, включающих в себя экземпляр и метод экземпляра, экземпляр для вызова есть экземпляр, содержащийся в вызываемй сущности.

7.6.6. Доступ к элементу массива

Доступ к элементу массива (element-access) состоит из *первичного-выражения-без-создания-массива*, за которым следует символ `"["`, далее следует *список-аргументов* и, наконец, символ `"]"`. *Список-аргументов* может состоять из одного или нескольких аргументов, разделенных запятыми.

```
доступ-к-элементу-массива:
   первичное-выражение-без-создания-массива [ список-аргументов ]
```

Список-аргументов не может содержать аргументы `ref` или `out`.

Доступ-к-элементу-массива является динамически связанным (раздел 7.2.2), если выполняется по меньшей мере один из следующих пунктов:

- *Первичное-выражение-без-создания-массива* имеет тип времени компиляции `dynamic`.
- По меньшей мере одно выражение *списка-аргументов* имеет тип времени компиляции `dynamic` и *первичное-выражение-без-создания-массива* не имеет тип массива.

В этом случае компилятор классифицирует *доступ-к-элементу-массива* как значение типа `dynamic`. Приведенные ниже правила, определяющие значение *доступа-к-элементу-массива*, применяются затем во время выполнения программы, используя тип времени выполнения вместо типа времени компиляции для *первичного-выражения-без-создания-массива* и для выражений из *списка-аргументов*, которые имеют тип времени компиляции `dynamic`. Если тип времени компиляции *первичного-выражения-без-создания-массива* не является типом `dynamic`, то *доступ-к-элементу-массива* частично проверяется во время компиляции, как описано в разделе 7.5.4.

Если *первичное-выражение-без-создания-массива* для *доступа-к-элементу-массива* является значением *типа-массива*, то *доступ-к-элементу-массива* есть доступ к массиву (раздел 7.6.6.1). В противном случае *первичное-выражение-без-создания-массива* должно быть переменной или значением типа класса, структуры или интерфейса, которые имеют один индексатор или более, и в этом случае *доступ-к-элементу-массива* является доступом к индексатору (раздел 7.6.6.2).

7.6.6.1. Доступ к массиву

В случае доступа к массиву *первичное-выражение-без-создания-массива* для *доступа-к-элементу-массива* должно быть значением типа массива. Кроме того, *список-аргументов* доступа к массиву не может содержать именованных аргументов. Количество выражений в *списке-аргументов* должно быть таким же, как размерность массива, и каждое выражение должно быть типа `int`, `uint`, `long` или `ulong` или должно быть неявно приводимым к одному или более из этих типов.

Результат вычисления доступа к массиву есть переменная типа элемента массива, а именно элемент массива, выбранный при помощи значения выражения (или значений выражений) в списке аргументов.

> **ДЖОН СКИТ**
>
> Тот факт, что результатом является переменная, важен: это означает, что при вызове метода вы можете использовать элементы массива как аргументы `ref` или `out`. Учитывая ковариантность массива, *фактический* тип ячейки хранения проверяется до вызова метода.

Обработка во время выполнения программы доступа к массиву в форме `P[A]`, где `P` является *первичным-выражением-без-создания-массива* для *типа-массива* и `A` является *списком-аргументов*, включает в себя следующие шаги:

- Вычисляется `P`. Если это приводит к исключению, дальнейшие шаги не выполняются.
- Выражения индекса для *списка-аргументов* вычисляются по порядку слева направо. Далее для каждого индексного выражения выполняется приведение к одному из следующих типов: `int`, `uint`, `long`, `ulong`. Будет выбран первый из этих типов, для которого существует неявное приведение (раздел 6.1). Например, если выражение индекса имеет тип `short`, то выполняется неявное приведение к типу `int`, поскольку возможны неявные преобразования из `short` к `int` и из `short` к `long`. Если вычисление выражения индекса или последующее неявное преобразование приводят к исключению, то следующие выражения индекса не вычисляются и никакие дальнейшие шаги не выполняются.
- Значение `P` проверяется на допустимость. Если значение `P` есть `null`, выбрасывается исключение `System.NullReferenceException` и никакие дальнейшие шаги не выполняются.
- Значение каждого выражения из *списка-аргументов* проверяется на соответствие фактическим границам каждого измерения экземпляра массива, на который ссылается `P`. Если одно или более значений выходят за границы, выбрасывается исключение `System.IndexOutOfRangeException` и дальнейшие шаги не выполняются.
- Вычисляется местоположение элемента массива, заданное выражениями индекса, и это местоположение становится результатом доступа к массиву.

7.6.6.2. Доступ к индексатору

Для доступа к индексатору *первичное-выражение-без-создания-массива* для *доступа-к-элементу-массива* должно быть переменной или значением типа класса, структуры или интерфейса, и в этом типе должен быть реализован один или более индексатор, которые являются подходящими для *списка-аргументов доступа-к-элементу-массива*.

Обработка доступа во время связывания к индексатору в форме `P[A]`, где `P` является *первичным-выражением-без-создания-массива* класса, структуры или интерфейса `T` и `A` является *списком-аргументов*, состоит из следующих шагов:

- Создается набор индексаторов из `T`. Набор состоит из всех индексаторов, объявленных в `T` или в его родительском типе, которые не объявлены как `override` и достижимы в текущем контексте (раздел 3.5).
- Набор содержит только те индексаторы, которые достижимы и не скрыты другими индексаторами. Следующие правила применимы к каждому индексатору `S.I` в наборе, где `S` является типом, в котором объявлен индексатор `I`:
  - Если `I` не является подходящим для `A` (раздел 7.5.3.1), то `I` удаляется из набора.

- Если `I` является подходящим для `A` (раздел 7.5.3.1), то все индексаторы, объявленные в базовом типе для типа `S`, удаляются из набора.
- Если `I` является подходящим для `A` (раздел 7.5.3.1) и `S` есть тип класса, отличный от `object`, все индексаторы, объявленные в интерфейсе, удаляются из набора.

- Если в результате получился пустой набор кандидатов, то не существует подходящих интерфейсов и выдается ошибка времени связывания.
- Наилучший индексатор в наборе возможных индексаторов определяется с помощью правил разрешения перегрузки раздела 7.5.3. Если не может быть выбран единственный наилучший индексатор, то доступ к индексатору неоднозначен и выдается ошибка времени связывания.
- Вычисляются выражения индекса в *списке-аргументов* по порядку, слева направо. Результатом обработки доступа к индексатору является выражение, классифицируемое как доступ к индексатору. Выражение доступа к индексатору ссылается на индексатор, определенный на предшествующем шаге, и имеет связанные с ним выражение экземпляра `P` и список аргументов `A`.

В зависимости от контекста, в котором он используется, доступ к индексатору вызывает либо код доступа *get-accessor*, либо код доступа *set-accessor*. Если доступ к индексатору имеет целью назначение, вызывается *set-accessor*, чтобы присвоить новое значение (раздел 7.17.1). Во всех других случаях вызывается *get-accessor* для получения текущего значения (раздел 7.1.1).

ВЛАДИМИР РЕШЕТНИКОВ

Если соответствующий код доступа отсутствует или недостижим, выдается ошибка компиляции.

7.6.7. this-доступ

this-доступ состоит из зарезервированного слова `this`.

```
this-доступ:
    this
```

this-доступ разрешен только в *блоке* конструктора экземпляра, метода экземпляра или кода доступа к экземпляру. Он имеет одно из следующих значений:

- Когда `this` используется в *первичном-выражении* внутри конструктора экземпляра класса, оно классифицируется как значение. Типом этого значения будет тип экземпляра класса, внутри которого оно используется, и это значение является ссылкой на сконструированный объект.
- Когда `this` используется в *первичном-выражении* внутри метода экземпляра или кода доступа к экземпляру класса, оно классифицируется как значение. Типом значения будет тип экземпляра класса (раздел 10.3.1), внутри которого оно используется, и это значение является ссылкой на объект, для которого вызывается метод или код доступа.

- Когда `this` используется в *первичном-выражении* внутри конструктора экземпляра структуры, оно классифицируется как переменная. Типом переменной является тип экземпляра структуры, в которой оно используется, и эта переменная представляет создаваемую структуру. Переменная `this` конструктора экземпляра ведет себя в точности так же, как параметр `out` структурного типа — в частности, это означает, что переменная должна быть определенно присвоена в каждом пути выполнения конструктора экземпляра.
- Когда `this` используется в *первичном-выражении* внутри метода экземпляра или кода доступа к экземпляру структуры, оно классифицируется как переменная. Типом переменной является тип экземпляра структуры, внутри которой оно используется.
  - Если метод или код доступа не является итератором (раздел 10.14), переменная `this` представляет структуру, для которой вызывается метод или код доступа, и ведет себя в точности так же, как параметр `ref` структурного типа.
  - Если метод или код доступа является итератором, переменная `this` представляет копию структуры, для которой вызывается метод или код доступа, и ее поведение в точности такое же, как у *параметра-значения* структурного типа.

ДЖОН СКИТ

Возможность написать

```
this = new CustomStruct(...);
```

в методе внутри типа-значения всегда казалась мне глубоко ошибочной. Учитывая, что структуры, прежде всего, всегда должны быть неизменяемыми, я удивляюсь, как много упоминаний о допустимости использования подобного кода можно найти во всемирном своде законов для кода C#.

Использование `this` в первичных выражениях в ином контексте, чем изложено выше, приводит к ошибке компиляции. В частности, недопустимо ссылаться на `this` в статическом методе, в коде доступа к статическому свойству или в инициализаторе переменной при объявлении поля.

7.6.8. Base-доступ

Base-доступ включает в себя зарезервированное слово `base`, за которым следует либо символ «`.`» и идентификатор, либо *список-аргументов* в квадратных скобках:

```
base-доступ:
   base  .  идентификатор
   base  [ список-аргументов ]
```

Base-доступ используется для доступа к элементам базового класса, которые скрыты элементами текущего класса или структуры с теми же именами. *Base-доступ* раз-

решен только в *блоке* конструктора экземпляра, метода экземпляра или кода доступа к экземпляру. Когда в классе или структуре используется `base.I`, `I` должен обозначать элемент базового класса для этого класса или структуры. Когда в классе используется `base[E]`, в базовом классе должен существовать подходящий индексатор.

Во время связывания выражения *base-доступа* в форме `base.I` и `base[E]` вычисляются в точности так же, как если бы они были записаны как `((B)this).I` и `((B)this)[E]`, где `B` является базовым классом для класса или структуры, в которых эта конструкция используется. Таким образом, `base.I` и `base[E]` соответствуют `this.I` и `this[E]`, за исключением того, что `this` рассматривается как экземпляр базового класса.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Если base-доступ в последней форме диспетчеризуется динамически (то есть имеет аргумент типа `dynamic`), то выдается ошибка компиляции (CS1972).

Когда *base-доступ* ссылается на виртуальный функциональный элемент (метод, свойство или индексатор), это изменяет то, какой функциональный элемент будет вызван во время выполнения программы (раздел 7.5.4). Вызываемый функциональный элемент определяется нахождением наиболее производной реализации (раздел 10.6.3) функционального элемента в соответствии с `B` (а не в соответствии с фактическим типом времени `this`, как это происходит при не-базовом доступе). Таким образом, при переопределении виртуального функционального элемента *base-доступ* может быть использован для вызова унаследованной реализации функционального элемента. Если функциональный элемент, на который ссылается *base-доступ*, является абстрактным, выдается ошибка времени связывания.

> **ЭРИК ЛИППЕРТ**
>
> В компиляторе Microsoft C# версии 2.0 и выше код, сгенерированный для базовых вызовов виртуальных методов, выглядит как код для невиртуальных вызовов *определенного* метода, известного во время компиляции для базового класса. Если вы хотите «обмануть» компилятор, поменяв в новой версии библиотеки, в которой существует новое виртуальное переопределение «в серединке» без перекомпиляции кода, который выполняет базовый вызов, такой код будет продолжать вызывать *определенный* метод, заданный во время компиляции. Короче говоря, *base-вызовы не используют виртуальную диспетчеризацию*.

7.6.9. Постфиксные инкрементные и декрементные операции

```
пост-инкрементное-выражение:
   первичное-выражение ++

пост-декрементное-выражение:
   первичное-выражение --
```

Операнды постфиксных инкрементных или декрементных операций должны быть выражением, классифицируемым как переменная, доступ к свойству или доступ к индексатору. Результатом такой операции является значение того же типа, что и операнд.

Если *первичное-выражение* имеет тип времени компиляции `dynamic`, то операция является динамически связанной (раздел 7.2.2), *пост-инкрементное-выражение* и *пост-декрементное-выражение* имеют тип времени компиляции `dynamic`, и во время выполнения программы применяются следующие правила, использующие фактический тип *первичного-выражения*.

Если операнд постфиксной инкрементной или декрементной операции является доступом к свойству или индексатору, свойство или индексатор должны иметь оба коды доступа — `get` и `set`. Если это не так, выдается ошибка времени связывания.

Для выбора определенной реализации операции применяется унарная операция разрешения перегрузки (раздел 7.3.3). Предопределенные операции `++` и `--` существуют для следующих типов: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` и для перечислимого типа. Предопределенные операции `++` возвращают значение, которое получается прибавлением 1 к операнду, а предопределенные операции `--` возвращают значение, которое получается вычитанием 1 из операнда. В проверяемом контексте, если результат этого сложения или вычитания выходит за границы значений типа результата и тип результата есть целый тип или перечисление, выбрасывается исключение `System.OverflowException`.

Обработка во время выполнения программы постфиксной инкрементной или декрементной операции в форме `x++` или `x--` состоит из следующих шагов:

- Если `x` классифицируется как переменная:
  - Вычисляется `x`, чтобы получить переменную.
  - Значение `x` сохраняется.
  - Вызывается выбранная операция с сохраненным значением `x`.
  - Значение, возвращенное операцией, сохраняется в том месте, которое определено вычислением `x`.
  - Сохраненное значение `x` становится результатом операции.
- Если `x` классифицируется как доступ к свойству или к индексатору:
  - Вычисляются выражение экземпляра (если `x` не является `static`) и список аргументов (если `x` является доступом к индексатору), связанные с `x`, и результат используется в последующих вызовах кодов доступа `get` и `set`.
  - Вызывается код доступа `get` для `x` и возвращенное значение сохраняется.
  - Вызывается выбранная операция с сохраненным значением `x` в качестве аргумента.
  - Вызывается код доступа `set` для `x` со значением, возвращенным операцией в качестве его аргумента `value`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Если либо *код-доступа-get*, либо *код-доступа-set* отсутствует или недостижим, выдается ошибка компиляции.

- Сохраненное значение `x` становится результатом операции.

Операции `++` и `--` также поддерживают префиксную запись (раздел 7.7.5). Обычно результатом `x++` или `x--` является значение `x` перед операцией, а как результатом `++x` или `--x` является значение `x` после операции. В любом случае после операции `x` имеет то же самое значение.

Реализация операции `++` и операции `--` может быть вызвана как с помощью постфиксной нотации, так и с помощью префиксной нотации. Не могут существовать две отдельные реализации операции для двух нотаций.

7.6.10. Операция new

Операция `new` служит для создания новых экземпляров типов.

Существует три формы выражений `new`:

- Выражения создания объекта используются для создания новых экземпляров классов и типов-значений.
- Выражения создания массива используются для создания новых экземпляров массивов.
- Выражения создания делегатов используются для создания новых экземпляров делегатов.

Действие операции `new` предполагает создание экземпляра типа, но не обязательно предполагает динамическое распределение памяти. В частности, экземпляры типов-значений не требуют дополнительной памяти, кроме переменных, в которых они размещаются: когда операция `new` используется для создания экземпляров типов-значений, динамического распределения не происходит.

7.6.10.1. Выражения создания объекта

Выражение-создания-объекта используется для создания нового *экземпляра-класса* или *типа-значения*.

```
выражение-создания-объекта:
    new  тип  (  список-аргументов_opt  )  инициализатор-объекта-или-коллекции_opt
    new  тип  инициализатор-объекта-или-коллекции

инициализатор-объекта-или-коллекции:
    инициализатор-объекта
    инициализатор-коллекции
```

Тип выражения-создания-объекта должен быть *типом-класса*, *типом-значением* или *параметром-типом*. Этот тип не может быть `abstract`.

Необязательный *список-аргументов* (раздел 7.5.1) разрешен, только если этот *тип* является *классм* или *структурой*.

В выражении создания объекта может быть опущен список аргументов конструктора и круглые скобки, в которые он заключен, при условии, что он содержит инициализатор объекта или инициализатор коллекции. Отсутствие списка аргументов и круглых скобок эквивалентно пустому списку аргументов.

Обработка выражения создания объекта, которое включает в себя инициализатор объекта или инициализатор коллекции, состоит из следующих операций: сначала происходит обработка конструктора экземпляра, а затем обрабатываются инициализации элементов класса или элементов, определенных инициализатором объекта (раздел 7.6.10.2) или инициализатором коллекции (раздел 7.6.10.3).

Если любой из аргументов необязательного *списка-аргументов* имеет тип времени компиляции `dynamic`, то *выражение-создания-объекта* является динамически связанным (раздел 7.2.2), и во время выполнения программы применяются следующие правила, использующие фактический тип тех аргументов из *списка-аргументов*, которые имеют тип времени компиляции `dynamic`. Однако создание объекта подлежит ограниченной проверке во время компиляции, как описано в разделе 7.5.4.

Обработка во время связывания *выражения-создания-объекта* в форме `new T(A)`, где `T` является *типом-класса* или *типом-значением* и `A` является необязательным *списком-аргументов*, состоит из следующих шагов:

- Если `T` является *типом-значением* и `A` отсутствует:
  - *Выражение-создания-объекта* является вызовом конструктора по умолчанию. Результатом *выражения-создания-объекта* является значение типа `T` — а именно, значение по умолчанию для `T`, определяемое, как описано в разделе 4.1.1.
- Иначе, если `T` является *параметром типа* и `A` отсутствует:
  - Если не определены никакие ограничения типа-значения или конструктора (раздел 10.1.5) для `T`, выдается ошибка времени связывания.
  - Результатом *выражения-создания-объекта* является значение фактического типа, с которым связан параметр-тип, а именно результат вызова конструктора по умолчанию для данного типа. Фактический тип может быть ссылочным или типом-значением.
- Иначе, если `T` является *классом* или *структурой*:
  - Если `T` является *классом* `abstract`, выдается ошибка компиляции.
  - Конструктор экземпляра, который будет вызван, определяется с помощью правил разрешения перегрузки из раздела 7.5.3. Набор кандидатов конструкторов экземпляра состоит из всех достижимых конструкторов экземпляра, объявленных в `T`, которые применимы для `A` (раздел 7.5.3.1). Если набор кандидатов конструкторов экземпляра пустой или если единственный наилучший экземпляр конструктора не может быть выбран, выдается ошибка времени связывания.
  - Результатом *выражения-создания-объекта* является значение типа `T`, а именно значение, полученное при помощи вызова конструктора экземпляра, определенного на предшествующем шаге.

- Иначе *выражение-создания-объекта* является недопустимым и выдается ошибка времени связывания.

Даже если *выражение-создания-объекта* является динамически связанным, типом времени компиляции все еще является тип `T`.

Обработка во время выполнения программы *выражения-создания-объекта* в форме `new T(A)`, где `T` является классом или структурой и `A` является необязательным списком аргументов, состоит из следующих шагов:

- Если `T` является *классом*:
  - Размещается новый экземпляр класса `T`. Если памяти для размещения нового экземпляра недостаточно, выбрасывается исключение `System.OutOfMemoryException` и дальнейшие шаги не выполняются.
  - Все поля нового экземпляра инициализируются к их значениям по умолчанию (раздел 5.2).
  - Вызывается конструктор экземпляра в соответствии с правилами вызова функционального элемента (раздел 7.5.4). Ссылка на размещенный экземпляр автоматически передается конструктору экземпляра, и этот экземпляр может быть доступен из конструктора экземпляра как `this`.
- Если `T` является *структуой*:
  - Экземпляр типа `T` создается размещением временной локальной переменной. Так как для конструктора экземпляра *структуры* требуется явно присвоить значение каждого поля создаваемого экземпляра, инициализация временной переменной не является необходимой.
  - Конструктор экземпляра вызывается в соответствии с правилами вызова функционального элемента (раздел 7.5.4). Ссылка на размещенный экземпляр автоматически передается конструктору экземпляра, и экземпляр может быть доступен из конструктора как `this`.

7.6.10.2. Инициализаторы объектов

Инициализатор объекта определяет значения для нуля или более полей или свойств объекта.

```
инициализатор-объекта:
   {  список-инициализаторов-элементов_opt   }
   {  список-инициализаторов-элементов  ,  }

список-инициализаторов-элементов:
   инициализатор-элемента
   список-инициализаторов-элементов  ,  инициализатор-элемента

инициализатор-элемента:
   идентификатор  =  значение-инициализатора

значение-инициализатора:
   выражение
   инициализатор-объекта-или-коллекции
```

Инициализатор объекта состоит из последовательности инициализаторов элементов, заключенных в фигурные скобки и разделенные запятыми. Каждый инициализатор элемента должен содержать имя доступного поля или свойства инициализируемого объекта, за которым следует знак равенства и выражение, инициализатор объекта или инициализатор коллекции. Если инициализатор объекта содержит более одного инициализатора элемента для каждого поля или свойства, это является ошибкой. Для инициализатора объекта также недопустимо ссылаться на создаваемый инициализируемый объект.

Инициализатор элемента, определяющий выражение после знака равенства, обрабатывается так же, как присваивание для поля или свойства (раздел 7.17.1).

Инициализатор элемента, определяющий выражение после знака равенства, является **вложенным инициализатором объекта**, то есть инициализатором вложенного объекта. Вместо присваивания нового значения полю или свойству, присваивания в инициализаторе вложенного объекта обрабатываются как присваивания значения элементам поля или свойства. Инициализаторы вложенных объектов не могут применяться к свойствам типа-значения и к полям типа-значения, предназначенным только для чтения.

Инициализатор элемента, определяющий инициализатор коллекции после знака равенства, является инициализатором вложенной коллекции. Вместо присваивания новой коллекции полю или свойству, элементы, заданные в инициализаторе, добавляются к коллекции, на которую ссылается поле или свойство. Поле или свойство должно иметь тип коллекции, удовлетворяющий требованиям раздела 7.6.10.3.

В следующем примере класс представляет точку с двумя координатами:

```
public class Point
{
   int x, y;
   public int X { get { return x; } set { x = value; } }
   public int Y { get { return y; } set { y = value; } }
}
```

Экземпляр `Point` может быть создан и инициализирован следующим образом:

```
Point a = new Point { X = 0, Y = 1 };
```

что равнозначно

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

где `__a` является невидимой и недоступной другими способами временной переменной. Следующий класс представляет прямоугольник, построенный по двум точкам:

```
public class Rectangle
{
Point p1, p2;

   public Point P1 { get { return p1; } set { p1 = value; } }
   public Point P2 { get { return p2; } set { p2 = value; } }
}
```

Экземпляр `Rectangle` может быть создан и инициализирован следующим образом:

```
Rectangle r = new Rectangle {
   P1 = new Point { X = 0, Y = 1 },
   P2 = new Point { X = 2, Y = 3 }
};
```

что эквивалентно

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

где `__r`, `__p1` и `__p2` — временные переменные, которые другими способами невидимы и недостижимы.

ДЖОЗЕФ АЛЬБАХАРИ

Использование скрытых временных переменных полностью исключает возможность завершения при частично инициализированном объекте, даже если во время инициализации возникнет исключение. Вместо этого конструируемый объект полностью удаляется:

```
Point p = null;
int zero = 0;
try { p = new Point { X = 3, Y = 4 / zero }; }
         // Выбрасывается DivideByZeroException
catch { Console.WriteLine (p == null); }
         // Правильно
```

ЭРИК ЛИППЕРТ

Использование скрытой временной переменной также проясняет правила определенного присваивания. Вторая строка не эквивалентна первой:

```
Point p1 = new Point(); p1.Y = p1.X; // Верно
Point p2 = new Point() { Y = p2.X }; // Не верно; p2 пока не присвоено
```

Если конструктор `Rectangle` размещает два вложенных экземпляра `Point`:

```
public class Rectangle
{
   Point p1 = new Point();
   Point p2 = new Point();
   public Point P1 { get { return p1; } }
   public Point P2 { get { return p2; } }
}
```

Для инициализации вложенных экземпляров `Point` вместо присваивания новых экземпляров может быть использована следующая конструкция:

```
Rectangle r = new Rectangle {
   P1 = { X = 0, Y = 1 },
   P2 = { X = 2, Y = 3 }
};
```

что равнозначно

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

7.6.10.3. Инициализаторы коллекций

Инициализатор коллекции определяет элементы коллекции.

инициализатор-коллекции:
   { *список-инициализаторов-элементов* }
   { *список-инициализаторов-элементов* , }

список-инициализаторов-элементов:
   инициализатор-элемента
   список-инициализаторов-элементов , *инициализатор-элемента*

инициализатор-элемента:
   выражение-без-присваивания
   { *список-выражений* }

список-выражений:
   выражение
   список-выражений , *выражение*

Инициализатор коллекции состоит из последовательности инициализаторов элементов, заключенных в фигурные скобки и разделенных запятыми. Каждый инициализатор элемента определяет элемент, который добавляется к объекту (коллекции), который инициализируется, и состоит из списка выражений, заключенных в фигурные скобки и разделенных запятыми. Инициализатор элемента с единственным выражением может быть записан без скобок, но потом не может быть выражением присваивания значения во избежание неоднозначности инициализаторов элементов. Правила для *выражения-без-присваивания* определены в разделе 7.18.

Следующий пример выражения создания объекта включает в себя инициализатор коллекции:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Коллекция, к которой применяется инициализатор, должна иметь тип, реализующий `System.Collections.IEnumerable`, в противном случае выдается ошибка компиляции. Для каждого заданного элемента по порядку инициализатор коллекции вызывает метод `Add` для целевого объекта со списком выражений инициализатора элемента в качестве списка аргументов, применяя нормальное разрешение перегрузки для каждого вызова. Таким образом, объект (коллекция) должен содержать подходящие методы `Add` для каждого инициализатора элемента.

> **ЭРИК ЛИППЕРТ**
>
> Это правило кажется немного странным: инициализатор коллекции действует только когда объект реализует `IEnumerable` и имеет метод `Add`. Заметьте, что вы никогда не вызываете метод `IEnumerable` в инициализаторе коллеции! Так почему он требуется? Команда создателей C# провела исследование существующих объектов и сделала следующие открытия. Во-первых, все почти объекты, имеющие метод `Add`, либо являются коллекциями, либо выполняют некоторые арифметические операции. Мы явно не хотели, чтобы эта структура служила для арифметики — только для создания коллекций. Во-вторых, для объектов, являющихся коллекциями, нет ни одного общего интерфейса с методом `Add`, который они все бы реализовывали. В-третьих, все коллекции реализуют `IEnumerable`, но его не реализуют арифметические объекты. В силу всех этих причин мы решили использовать существование `IEnumerable` наряду с методом `Add` в качестве пробного камня для определения допустимости инициализатора коллекции.

Следующий класс представляет контакт с именем и списком телефонных номеров:

```
public class Contact
{
   string name;
   List<string> phoneNumbers = new List<string>();
   public string Name { get { return name; } set { name = value; } }
   public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

`List<Contact>` может быть создан и инициализирован следующим образом:

```
var contacts = new List<Contact> {
   new Contact {
      Name = "Chris Smith",
      PhoneNumbers = { "206-555-0101", "425-882-8080" }
   },
   new Contact {
      Name = "Bob Harris",
      PhoneNumbers = { "650-555-0199" }
   }
};
```

что равнозначно

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

где `__clist`, `__c1` и `__c2` являются временными переменными, которые невидимы и недостижимы другими способами.

7.6.10.4. Выражения создания массива

Выражение создания массива используется для создания нового экземпляра типа массива.

```
выражение-создания-массива:
   new  тип-не-массива  [ список-выражений ]  размерности_opt инициализатор-массива_opt
   new  тип-массива     инициализатор-массива
   new  размерность     инициализатор-массива
```

Выражение создания массива в первой форме размещает экземпляр массива типа, который получается путем удаления каждого из отдельных выражений из списка выражений. Например, выражение создания массива `new int[10, 20]` дает экземпляр массива типа `int[,]`. Каждое выражение из списка выражений должно быть типа `int`, `uint`, `long` или `ulong` или должно быть неявно приводимо к одному или нескольким типам из этого списка. Значение каждого выражения определяет длину соответствующей размерности в размещаемом экземпляре массива. Поскольку длина любой размерности массива должна быть неотрицательной, то если в списке выражений имеется *константное-выражение* с отрицательным значением, выдается ошибка компиляции.

Способ размещения элементов массива не задается, кроме как в небезопасном контексте (раздел 18.1).

Если выражение создания массива в первой форме включает в себя инициализатор массива, то каждое выражение в списке выражений должно быть константой, и размерность и длины размерностей, определенные в списке выражений, должны соответствовать аналогичным величинам в инициализаторе массива.

В выражении создания массива во второй или третьей форме размерность заданного типа массива должна соответствовать аналогичной величине в инициализаторе массива. Отдельные длины размерностей выводятся из числа элементов в каждом из соответствующих уровней вложенности в инициализаторе массива. Таким образом, выражение

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

в точности соответствует

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

Выражение создания массива в третьей форме ссылается на так называемое **выражение создания массива с неявно заданным типом**. Оно подобно второй форме за исключением того, что тип элементов массива не задан явно, но определяется как лучший общий тип (раздел 7.5.2.14) для набора выражений в инициализаторе массива. Для многомерных массивов, то есть для тех, *размерность* которых содержит по меньшей мере одну запятую, такой набор включает в себя все *выражения*, найденные во вложенных *инициализаторах-массива*.

Инициализаторы массива описаны далее в разделе 12.6.

Результат вычисления выражения создания массива классифицируется как значение, а именно как ссылка на размещаемый экземпляр массива. Обработка выражения создания массива во время выполнения программы состоит из следующих шагов:

- Вычисляются по порядку, слева направо, выражения для длин размерностей в списке выражений. После вычисления каждого выражения выполняется

неявное приведение (раздел 6.1) к одному из следующих типов: `int`, `uint`, `long`, `ulong`. Выбирается первый тип из этого списка, для которого существует такое неявное приведение. Если вычисление выражения или последующее неявное приведение порождают исключение, то следующие выражения не вычисляются и дальнейшие шаги не выполняются.

- Вычисленные значения для длин размерностей проверяются следующим образом. Если одно или несколько значений меньше нуля, выбрасывается исключение `System.OverflowException` и никакие дальнейшие шаги не выполняются.
- Размещается экземпляр массива с данными длинами размерностей. Если памяти для размещения нового экземпляра недостаточно, выбрасывается исключение `System.OutOfMemoryException` и дальнейшие шаги не выполняются.
- Все элементы нового массива инициализируются их значениями по умолчанию (раздел 5.2).
- Если выражение создания массива содержит инициализатор массива, то вычисляется каждое выражение в инициализаторе массива и его значение присваивается соответствующему элементу массива. Вычисления и присваивания выполняются в том порядке, в котором выражения записаны в инициализаторе массива — иными словами, элементы инициализируются в порядке возрастания индекса, начиная с возрастания индекса самой правой размерности. Если вычисление данного выражения или последующее присваивание значения элементу массива приводит к исключению, то никакие следующие элементы не инициализируются (и оставшиеся элементы будут, таким образом, иметь значения по умолчанию).

Выражение создания массива позволяет создать массив с элементами типа массива, но элементы такого массива должны быть инициализированы вручную. Например, оператор

```
int[][] a = new int[100][];
```

создает одномерный массив из 100 элементов типа `int[]`. Начальное значение каждого элемента есть `null`. Невозможно, чтобы то же самое выражение создания массива также инициализировало значения подмассивов, и оператор

```
int[][] a = new int[100][5]; // Ошибка
```

приводит к появлению ошибки компиляции. Вместо этого значения элементов подмассивов должны быть инициализированы вручную, например, следующим образом:

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

БИЛЛ ВАГНЕР

Комментарий, приведенный ниже, нарушает правило FxCop, но объясняет, при каких условиях вы можете нарушить это правило.

Когда массив массивов имеет «прямоугольную» форму, то есть, когда все подмассивы имеют одну и ту же длину, более эффективно использовать многомерный

массив. В примере, приведенном выше, реализация массива массивов создает 101 объект — один внешний массив и 100 подмассивов. Напротив, запись

```
int[,] = new int[100, 5];
```

создает только один объект — двумерный массив, и размещение выполняется одним оператором.

> **ПИТЕР СЕСТОФТ**
>
> Довольно сомнительно, когда спецификация языка содержит утверждения об эффективности, когда дело касается реализации, а не семантики. Действительно, может быть быстрее разместить единственный блок из 500 целых чисел, чем 100 блоков из 5 целых чисел (и затем управлять им), но алгоритм матричного умножения, работающий на «медленном» представлении массива массивов, может оказаться быстрее, чем такой же алгоритм, работающий на «эффективном» представлении прямоугольного массива — возможно, благодаря «умному» JIT-компилятору и современной архитектуре CPU.

Ниже приведены примеры выражений создания массива с неявно заданным типом:

```
var a = new[] { 1, 10, 100, 1000 };                    // int[]
var b = new[] { 1, 1.5, 2, 2.5 };                      // double[]
var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]
var d = new[] { 1, "one", 2, "two" };                  // Error
```

Последнее выражение приводит к ошибке компиляции, поскольку ни `int`, ни `string` не могут быть неявно преобразованы к другому типу, так что в данном случае нет лучшего общего типа. В данном случае можно использовать выражение создания массива с явно заданным типом — например, определив тип как `object[]`. В качестве альтернативы один из элементов может быть приведен к общему базовому типу, который затем становится выведенным типом элемента.

Выражения создания массива с неявно заданным типом можно комбинировать с инициализаторами анонимных объектов (раздел 7.6.10.6) для создания структур данных анонимного типа. Например:

```
var contacts = new[] {
   new {
      Name = "Chris Smith",
      PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
   },
   new {
      Name = "Bob Harris",
      PhoneNumbers = new[] { "650-555-0199" }
   }
};
```

7.6.10.5. Выражения создания делегата

Выражение создания делегата используется для создания нового экземпляра *типа-делегата*.

выражение-создания-делегата:
   new *тип-делегата* **(** *выражение* **)**

Аргументом выражения создания делегата должна быть группа методов, анонимная функция или значение типа времени компиляции `dynamic` либо *типа-делегата*. Если аргументом является группа методов, она идентифицирует метод, и для метода экземпляра объект, для которого создается делегат. Если аргументом является анонимная функция, она прямо определяет параметры и тело метода для целевого делегата. Если аргументом является значение, оно идентифицирует экземпляр делегата, для которого создается копия.

Если *выражение* имеет тип времени компиляции `dynamic`, *выражение-создания-делегат*а является динамически связанным (раздел 7.2.2) и правила, приведенные ниже, применяются во время выполнения программы и используют фактический тип *выражения*. В противном случае правила применяются во время компиляции.

Обработка во время связывания *выражения-создания-делегата* в форме `new D(E)`, где `D` является *типом-делегата* и `E` является *выражением*, состоит из следующих шагов:

- Если `E` является группой методов, то выражение создания делегата обрабатывается таким же способом, как преобразование группы методов (раздел 6.6) от `E` к `D`.
- Если `E` является анонимной функцией, выражение создания делегата обрабатывается таким же способом, как преобразование анонимной функции от `E` к `D` (раздел 6.5).
- Если `E` является значением, `E` должно быть совместимо с `D` (раздел 15.1) и результатом будет ссылка на создаваемый делегат типа `D`, который ссылается на тот же список вызова, что и `E`. Если `E` несовместим с `D`, выдается ошибка компиляции.

Обработка во время выполнения программы *выражения-создания-делегата* в форме `new D(E)`, где `D` является *типом-делегата* и `E` является *выражением*, состоит из следующих шагов:

- Если `E` является группой методов, выражение создания делегата вычисляется как преобразование группы методов (раздел 6.6) от `E` к `D`.
- Если `E` является анонимной функцией, создание делегата вычисляется как преобразование анонимной функции от `E` к `D` (раздел 6.5).
- Если `E` является значением *типа-делегата*:
  - Вычисляется `E`. Если это вычисление приводит к исключению, никакие дальнейшие шаги не выполняются.
  - Если значение `E` есть `null`, выбрасывается исключение `System.NullReferenceException` и дальнейшие шаги не выполняются.
  - Размещается новый экземпляр делегата типа `D`. Если памяти для размещения недостаточно, выбрасывается исключение `System.OutOfMemoryException` и дальнейшие шаги не выполняются.
  - Новый экземпляр делегата инициализируется тем же списком вызова, что у экземпляра делегата, заданного `E`.

Список вызова делегата определяется при инстанцировании значения делегата и затем остается неизменным в течение всего времени жизни делегата. Иными словами, невозможно изменить целевые вызываемые сущности созданного делегата. Когда два делегата комбинируются или один удаляется из другого (раздел 15.1), в результате появляется новый делегат; в существующих делегатах их содержимое изменить нельзя.

Невозможно создать делегат, который ссылается на свойство, индексатор, операцию, определенную пользователем, конструктор экземпляра, деструктор и статический конструктор.

Как было описано выше, когда делегат создается из группы методов, список формальных параметров и тип возвращаемого значения определяют, который из перегруженных методов будет выбран. В примере

```
delegate double DoubleFunc(double x);
class A
{
   DoubleFunc f = new DoubleFunc(Square);

   static float Square(float x) {
      return x * x;
   }
   static double Square(double x) {
      return x * x;
   }
}
```

поле `A.f` инициализируется делегатом, который ссылается на второй метод `Square`, поскольку этот метод в точности соответствует списку формальных параметров и возвращаемому типу `DoubleFunc`. Если бы второй метод отсутствовал, возникла бы ошибка компиляции.

7.6.10.6. Выражение создания анонимного объекта

Выражение создания анонимного объекта используется для создания объекта анонимного типа.

```
выражение-создания-анонимного-объекта:
   new   инициализатор-анонимного-объекта

инициализатор-анонимного-объекта:
   {  список-объявлений элементовopt  }
   {  список-объявлений-элементов  ,  }

список-объявлений-элементов:
   объявление-элемента
   список-объявлений-элементов   ,   объявление-элемента

объявление-элемента:
   простое-имя
   доступ-к-элементу
   base-доступ
   идентификатор = выражение
```

ВЛАДИМИР РЕШЕТНИКОВ

Base-доступ может являться *описателем элемента*, только если он существует в форме `base`.*идентификатор*, то есть никакой доступ к базовому индексатору здесь не разрешен.

Инициализатор анонимного объекта объявляет анонимный тип и возвращает экземпляр этого типа. Анонимный тип является безымянным типом класса, который наследуется непосредственно от `object`. Элементы анонимного типа представляют последовательность свойств только для чтения, выведенных из инициализатора анонимного объекта, использованного для создания экземпляра типа. В частности, инициализатор анонимного объекта в форме

```
new { p1 = e1 , p2 = e2 , ... pn = en }
```

объявляет анонимный тип в форме

```
class __Anonymous1
{
   private readonly T1 f1 ;
   private readonly T2 f2 ;
   ...
   private readonly Tn fn ;
   public __Anonymous1(T1 a1, T2 a2,..., Tn an) {
      f1 = a1 ;
      f2 = a2 ;
      ...
      fn = an ;
   }
   public T1 p1 { get { return f1 ; } }
   public T2 p2 { get { return f2 ; } }
   ...
   public Tn pn { get { return fn ; } }

   public override bool Equals(object o) { ... }
   public override int GetHashCode() { ... }
}
```

где каждый T_x является типом соответствующего выражения e_x. Выражение, используемое в *описателе-элемента*, должно иметь тип. Таким образом, если выражение в *описателе-элемента* имеет значение `null` или является анонимной функцией, выдается ошибка компиляции. Также приводит к ошибке компиляции выражение небезопасного типа.

ЭРИК ЛИППЕРТ

Фактический код, сгенерированный в реализации Microsoft для анонимного типа, несколько более сложен, чем предполагается в этом обсуждении, поскольку было сильное желание (о чем будет упомянуто позднее), чтобы структурно эквивалентные анонимные типы были представлены одним и тем же типом во всей программе. Поскольку типы полей могут быть защищенными (protected) вложенными типами, становится трудно вычислить, где именно сгенерировать анонимный класс, так чтобы он мог быть эффективно разделяться различными выведенными типами. По этой причине реализация Microsoft на самом деле создает обобщенный класс и затем конструирует его с аргументами подходящего типа.

Имя анонимного типа автоматически создается компилятором и на него не может быть ссылок в тексте программы.

> **ДЖОН СКИТ**
> Хотя это звучит парадоксально, я был бы счастлив увидеть именованные анонимные типы: классы, которые легко выразить как анонимные типы и которые обладают теми же свойствами неизменяемости, естественного равенства и безопасностью типов и имен свойств — но с именем. Хотя многие средства могут «расширять» анонимные типы в эквивалентные нормальным классы, при этом теряется краткость выражения типа.

Внутри одной и той же программы два инициализатора анонимных объекта, определяющие последовательность свойств с одинаковыми именами и типами времени компиляции в том же порядке, порождают экземпляры одного и того же анонимного типа.

В примере

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

присваивание в последней строке разрешено, поскольку `p1` и `p2` имеют один и тот же анонимный тип.

Методы `Equals` и `GetHashcode` для анонимных типов переопределяют методы, унаследованные от `object`, и определяются в терминах `Equals` и `GetHashcode` для свойств, так что два экземпляра одного и того же анонимного типа равны тогда и только тогда, когда все их свойства равны.

Описатель элемента может быть сокращен до простого имени (раздел 7.5.2), доступа к элементу (раздел 7.5.4) или базового доступа (раздел 7.6.8). Это называется **инициализатором проекции** (*projection initializer*) и является сокращенным обозначением объявления свойства с тем же именем и присваивания ему значения. В частности, описатель элемента в форме

идентификатор выражение **.** *идентификатор*

в точности эквивалентен следующему, соответственно:

идентификатор **=** *идентификатор идентификатор* **=** *выражение* **.** *идентификатор*

Таким образом, в инициализаторе проекции *идентификатор* выбирает как значение, так и поле или свойство, которому это значение присваивается. Интуитивно ясно, что инициализатор проекции описывает не только значение, но также и имя этого значения.

7.6.11. Операция typeof

Операция `typeof` используется для получения объекта `System.Type` для типа.

выражение-typeof:
  typeof **(** *тип* **)**
  typeof **(** *имя-неограниченного-типа* **)**
  typeof **(** void **)**

продолжение ⇨

```
имя-неограниченного-типа:
   идентификатор   описатель-обобщенной-размерности_opt
   идентификатор   ::   идентификатор   описатель-обобщенной-размерности_opt
   имя-неограниченного-типа   .   идентификатор   описатель-обобщенной-размерности_opt

описатель-обобщенной-размерности:
   <   запятые_opt   >

запятые:
   ,
   запятые   ,
```

Первая форма *выражения-typeof* состоит из ключевого слова `typeof`, за которым следует заключенный в скобки *тип*. Результатом выражения в этой форме является объект `System.Type` для указанного типа. Существует только один объект `System.Type` для любого заданного типа. Это означает, что для типа `T` `typeof(T) == typeof(T)` всегда верно. *Тип* не может быть типом `dynamic`.

Вторая форма *выражения-typeof* состоит из ключевого слова `typeof`, за которым следует *имя-неограниченного-типа*. *Имя-неограниченного-типа* очень похоже на *имя-типа* (раздел 3.8) за исключением того, что *имя-неограниченного-типа* содержит *описатели-обобщенных-размерностей*, тогда как *имя-типа* содержит *списки-аргументов-типов*. Когда операнд *выражения typeof* представляет собой последовательность символов, удовлетворяющую требованиям как грамматики *имени-неограниченного-типа*, так и грамматики *имени-типа* — а именно, когда он не содержит ни *описатель-обобщенных-размерностей*, ни *список-аргументов-типов*, — такая последовательность символов считается *именем-типа*. Значение имени неограниченного типа определяется следующим образом:

- Преобразуйте последовательность символов к *имени-типа*, заменив каждый *описатель-обобщенных-размерностей* на *список-аргументов-типов*, содержащий то же число запятых и ключевое слово `object` в качестве каждого аргумента-типа.
- Определите получившееся в результате имя типа, пока игнорируя все ограничения параметров-типов.
- *Имя-неограниченного-типа* разрешается для неограниченного обобщенного типа, который связывается с результирующим сконструированным типом (раздел 4.4.3).

Результатом *выражения-typeof* является объект `System.Type` для получившегося в результате неограниченного обобщенного типа.

Третья форма *выражения-typeof* состоит из ключевого слова `typeof`, за которым следует заключенное в скобки ключевое слово `void`. Результатом выражения в этой форме является объект `System.Type`, который обозначает отсутствие типа. Объект, возвращенный формой `typeof(void)`, отличается от всех других. Этот особый тип объекта полезен в библиотеках классов, которые позволяют выполнять рефлексию для методов языка, в котором для этих методов нужно представить тип любого возвращаемого значения метода, включая `void`, с помощью экземпляра `System.Type`.

Операция `typeof` может применяться к параметру-типу. Результатом будет объект `System.Type` для типа времени выполнения, который связан с параметром-типом. Операция `typeof` также может применяться к сконструированному типу и к неограниченному обобщенному типу (раздел 4.4.3). Объект `System.Type` для неограниченного обобщенного типа — не то же самое, что объект `System.Type` для типа экземпляра.

Тип экземпляра всегда является закрытым сконструированным типом во время выполнения программы, так что его объект `System.Type` зависит от используемых во время выполнения программы аргументов-типов, в то время как неограниченный обобщенный тип не имеет аргументов-типов.

Пример

```
using System;
class X<T>
{
   public static void PrintTypes() {
      Type[] t = {
         typeof(int),
         typeof(System.Int32),
         typeof(string),
         typeof(double[]),
         typeof(void),
         typeof(T),
         typeof(X<T>),
         typeof(X<X<T>>),
         typeof(X<>)
      };
      for (int i = 0; i < t.Length; i++) {
         Console.WriteLine(t[i]);
      }
   }
}
class Test
{
   static void Main() {
      X<int>.PrintTypes();
   }
}
```

дает следующий вывод:

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

Отметим, что `int` и `System.Int32` одного и того же типа.

Также отметим, что результат `typeof(X<>)` не зависит от типа-аргумента, а результат `typeof(X<T>)` — зависит.

7.6.12. Операции checked и unchecked

Операции `checked` и `unchecked` используются для управления *контекстом контроля переполнения* при целочисленных арифметических операциях и преобразованиях.

выражение-checked:
  checked **(** *выражение* **)**

выражение-unchecked:
  unchecked **(** *выражение* **)**

Операция `checked` вычисляет значение содержащегося в скобках выражения в проверяемом контексте, а операция `unchecked` вычисляет значение содержащегося в скобках выражения в непроверяемом контексте. *Выражение-checked* и *выражение-unchecked* в точности соответствуют *выражению-в-скобках* (раздел 7.6.3), за исключением того, что содержащееся в скобках выражение вычисляется в данном контексте контроля переполнения.

> **ДЖОН СКИТ**
>
> В большинстве случаев переполнение указывает на ошибку, но, как показывает мой опыт, большинство разработчиков (включая меня) обычно обходятся без включения проверки по умолчанию. Очевидно, что этот подход ведет к потере производительности, но он обычно предпочтителен для данных, которые могут быть искажены. Возможно, имело бы смысл использовать `/checked+` по умолчанию для отладки и `/checked-` по умолчанию для готовой версии, хотя я вообще осторожен в отношении различий в поведении в отладочной и выпускной версиях программы. Это немного напоминает поведение исключений для пересекающихся потоков в Windows Forms.
> Одна из ситуаций, в которых уместно отсутствие контроля переполнения, — в реализациях `GetHashCode`; в этом случае величина полученного значения не играет роли; по сути, оно является случайной комбинацией битов.

Проверка контекста контроля переполнения может также выполняться при помощи операторов `checked` и `unchecked` (раздел 8.11).

Проверка контекста контроля переполнения, организованная с помощью операций и операторов `checked` и `unchecked`, влияет на следующие операции:

- Предопределенные операции `++` и `--` (разделы 7.6.9 и 7.7.5), когда операнд имеет целочисленный тип.
- Предопределенные бинарные операции `+`, `-`, `*` и `/` (раздел 7.8), когда оба операнда имеют целочисленный тип.
- Явные приведения (раздел 6.2.1) из одного целочисленного типа к другому целочисленному типу или из типов `float` или `double` к целочисленному типу.

Когда одна из приведенных выше операций приводит к результату, который слишком велик, чтобы быть представленным в типе назначения, контекст, в котором операция выполнялась, определяет дальнейшие действия:

- В проверяемом контексте, если операция является константным выражением (раздел 7.19), выдается ошибка компиляции. В противном случае, когда опе-

рация выполняется в процессе работы программы, выбрасывается исключение `OverflowException`.

- В непроверяемом контексте отбрасываются старшие биты, которые не умещаются в тип назначения.

Для неконстантных выражений (выражений, которые вычисляются в процессе работы программы), которые не включены в область действия операций или операторов `checked` или `unchecked`, контекст контроля переполнения по умолчанию `unchecked`, до тех пор пока внешние факторы (такие как ключи компилятора и конфигурация окружения) не зададут `checked`.

Для константных выражений (выражений, которые полностью вычисляются во время компиляции) контекст контроля переполнения всегда `checked`. Пока константное выражение не помещено явно в контекст `unchecked`, переполнения, которые появятся в процессе вычислений во время компиляции, всегда приводят к ошибке компиляции.

Проверяемый или непроверяемый контекст не оказывает влияния на тело анонимной функции.

В примере:

```
class Test
{
   static readonly int x = 1000000;
   static readonly int y = 1000000;

   static int F() {
      return checked(x * y);        // Генерирует OverflowException
   }

   static int G() {
      return unchecked(x * y);      // Возвращает -727379968
   }

   static int H() {
      return x * y;                // Зависит от значений по умолчанию
   }
}
```

ошибки компиляции не выдаются, так как ни одно из выражений не может быть вычислено во время компиляции. Во время выполнения программы метод `F` генерирует исключение `System.OverflowException`, а метод `G` возвращает значение −727379968 (младшие 32 разряда результата, выходящего за границы интервала значений). Поведение метода `H` зависит от контекста контроля переполнения по умолчанию для компиляции: оно либо такое же, как у метода `F`, либо такое же, как у метода `G`.

В примере

```
class Test
{
   const int x = 1000000;
   const int y = 1000000;

   static int F() {
      return checked(x * y);            // Ошибка компиляции, переполнение
   }
```

продолжение ⇗

```
   static int G() {
      return unchecked(x * y);          // Возвращает -727379968
   }

   static int H() {
      return x * y;                     // Ошибка компиляции, переполнение
   }
}
```

переполнение, которое происходит при вычислении константных выражений в F и H, приводит к сообщению об ошибке компиляции, поскольку выражения вычисляются в проверяемом контексте. Переполнение также происходит при вычислении константных выражений в G, но так как вычисление выполняется в непроверяемом контексте, о переполнении не сообщается.

Операции checked и unchecked действуют на контекст контроля переполнения только операций, заключенных в круглые скобки. Эти операции не влияют на функциональные элементы, которые вызываются в результате вычисления содержащегося в скобках выражения. В примере

```
class Test
{
   static int Multiply(int x, int y) {
      return x * y;
   }
   static int F() {
      return checked(Multiply(1000000, 1000000));
   }
}
```

использование checked в F не влияет на вычисление x * y в Multiply, так что x * y вычисляется в контексте контроля переполнения, установленного по умолчанию.

Операция unchecked удобна для записи констант целочисленных типов со знаком в шестнадцатеричном представлении. Например:

```
class Test
{
   public const int AllBits = unchecked((int)0xFFFFFFFF);
   public const int HighBit = unchecked((int)0x80000000);
}
```

Обе шестнадцатеричные константы имеют тип uint. Поскольку они выходят за границы интервала значений int, в отсутствие операции unchecked приведения к int будут давать ошибку компиляции.

Операции и операции checked и unchecked позволяют программистам управлять некоторыми аспектами численных расчетов. Однако поведение некоторых численных операций зависит от типа данных их операндов. Например, при умножении двух чисел типа decimal переполнение всегда приводит к появлению исключения *даже* внутри явной конструкции unchecked. Умножение же двух чисел с плавающей точкой никогда не дает исключения при переполнении *даже* внутри явной конструкции checked. Добавим, что режим проверки *никогда* не влияет на другие операции, независимо от того, задан ли он явно или установлен по умолчанию.

7.6.13. Выражения значений по умолчанию

Выражение значения по умолчанию используется для получения значения по умолчанию (раздел 5.2) для типа. Обычно выражение значения по умолчанию используется для параметров-типов, поскольку может быть не известно, имеет ли параметр-тип ссылочный тип или тип-значение. (Не существует преобразования от литерала `null` к параметру-типу, если известно, что он ссылочного типа.)

выражение-значения-по-умолчанию:
  default (*тип* **)**

Если *тип* в *выражении-значения-по-умолчанию* при вычислении во время выполнения программы оказывается ссылочным типом, результатом является значение `null`, преобразованное к данному типу. Если *тип* в *выражении-значения-по-умолчанию* при вычислении во время выполнения программы оказывается типом-значением, то результатом будет значение по умолчанию этого *типа-значения* (раздел 4.1.2).

Выражение-значения-по-умолчанию является константным выражением (раздел 7.19), если тип является ссылочным или параметром-типом ссылочного типа (раздел 10.1.5). Добавим, что *выражение-значения-по-умолчанию* является константным выражением, если тип является одним из следующих типов-значений: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool` или перечислением.

ДЖОЗЕФ АЛЬБАХАРИ

Выражение значения по умолчанию особенно полезно для обобщенных параметров-типов. Реализация Microsoft LINQ расширяет использование этой конструкции, применяя ее в таких операциях, как `FirstOrDefault`, `SingleOrDefault` и `DefaultIfEmpty`.

7.6.14. Выражения анонимных методов

Выражение-анонимного-метода является одним из двух способов определения анонимной функции. Эти выражения описаны далее в разделе 7.15.

7.7. Унарные операции

Операции `+`, `-`, `!`, `~`, `++`, `--` и операции приведения называются унарными операциями.

унарное-выражение:
  первичное-выражение
  + *унарное-выражение*
  - *унарное-выражение*
  ! *унарное-выражение*
  ~ *унарное-выражение*
  пре-инкрементное-выражение
  пре-декрементное-выражение
  выражение-приведения

Если операнд *унарного выражения* имеет тип времени компиляции `dynamic`, он является динамически связанным (раздел 7.2.2). В этом случае типом времени компиляции *унарного-выражения* будет тип `dynamic`, и во время выполнения будет производиться разрешение с использованием типа операнда времени выполнения, описанное ниже.

7.7.1. Унарная операция «плюс»

Для операции в форме `+x` для выбора реализации операции применяется разрешение перегрузки унарной операции (раздел 7.3.3). Операнд преобразуется к типу параметра выбранной операции, и типом результата будет тип, возвращаемый операцией. Предопределенные унарные операции «плюс» приведены ниже:

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

Для каждой из этих операций результат есть просто значение операнда.

> **ЭРИК ЛИППЕРТ**
>
> Унарная операция «плюс» является наименее используемой операцией из всех. Она включена для полноты, так что вы можете написать `int x = -30; int y = +40;`, если хотите акцентировать внимание на том, что значение положительно.

7.7.2. Унарная операция «минус»

Для операции в форме `–x` для выбора реализации операции применяется разрешение перегрузки унарной операции (раздел 7.3.3). Операнд преобразуется к типу параметра выбранной операции, и типом результата будет тип, возвращаемый операцией. Предопределенные операции отрицания приведены ниже:

- Отрицание для целочисленных типов

```
int operator –(int x);
long operator –(long x);
```

Результат вычисляется путем вычитания `x` из нуля. Если значение `x` является наименьшим значением, которое может быть представлено данным типом операнда (-2^{31} для `int` или -2^{63} для `long`), то математическое отрицание `x` невозможно представить в данном типе операнда. Если такая ситуация возникает в проверяемом контексте, генерируется исключение `System.OverflowException`; если же она возникает в непроверяемом контексте, результатом будет значение операнда и о переполнении не будет сообщаться.

Если операнд операции отрицания имеет тип `uint`, он приводится к типу `long`, и тип результата будет `long`. Исключением является правило, которое позволяет

записать значение `int` –2147483648 (-2^{31}) в виде десятичного целого литерала (раздел 2.4.42).

Если операнд операции отрицания имеет тип `ulong`, выдается ошибка компиляции. Исключением является правило, которое позволяет записать значение `long` –9223372036854775808 (-2^{63}) в виде десятичного целого литерала (раздел 2.4.4.2).

- Отрицание для типов с плавающей точкой:

```
float operator –(float x);
double operator –(double x);
```

Результатом будет значение `x` с противоположным знаком. Если `x` есть NaN, результат также будет NaN.

- Отрицание для десятичного типа:

```
decimal operator –(decimal x);
```

Результат получается вычитанием `x` из нуля. Отрицание для десятичного типа эквивалентно использованию унарной операции минус для типа `System.Decimal`.

7.7.3. Логическая операция отрицания

Для операции в форме `!x` для выбора реализации операции применяется разрешение перегрузки унарной операции (раздел 7.3.3). Операнд преобразуется к типу параметра выбранной операции, и типом результата будет тип, возвращаемый операцией.

Существует только одна предопределенная логическая операция отрицания:

```
bool operator !(bool x);
```

Эта операция вычисляет логическое отрицание операнда: если операнд есть `true`, результатом будет `false`. Если операнд есть `false`, результатом будет `true`.

7.7.4. Операция поразрядного дополнения

Для операции в форме `~x` для выбора реализации операции применяется разрешение перегрузки унарной операции (раздел 7.3.3). Операнд приводится к типу параметра выбранной операции, и типом результата будет тип, возвращаемый операцией. Предопределенные операции поразрядного дополнения представлены ниже:

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

Результатом каждой из этих операций является поразрядное дополнение `x`.

Каждый перечислимый тип `E` неявно содержит следующую операцию поразрядного дополнения:

```
E operator ~(E x);
```

БИЛЛ ВАГНЕР

Эта операция часто создает недопустимое значение перечисления.

Результат вычисления `~x`, где `x` является выражением перечислимого типа `E`, имеющего базовый тип `U`, в точности такой же, как результат вычисления `(E)(~(U)x)`.

7.7.5. Префиксные инкрементные и декрементные операции

пре-инкрементное-выражение:
    `++` *унарное-выражение*

пре-декрементное-выражение:
    `--` *унарное-выражение*

Операнд префиксной инкрементной или декрементной операции должен быть выражением, классифицируемым как переменная, доступ к свойству или доступ к индексатору. Результатом операции является значение того же типа, что и у операнда.

Если операнд префиксной инкрементной или декрементной операции является доступом к свойству или к индексатору, свойство или индексатор должны иметь оба кода доступа (`get` и `set`). Если это не так, происходит ошибка компиляции.

Для выбора определенной реализации операции применяется разрешение перегрузки унарной операции (раздел 7.3.3). Предопределенные операции `++` и `--` существуют для следующих типов: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` и для любого перечислимого типа. Предопределенные операции `++` возвращают значение, полученное добавлением 1 к операнду, а предопределенные операции `--` возвращают значение, полученное вычитанием 1 из операнда. В проверяемом контексте, если типом результата является целочисленный или перечислимый тип, выбрасывается исключение `System.OverflowException`.

Обработка во время выполнения программы префиксной инкрементной или декрементной операции в форме `++x` или `--x` состоит из следующих шагов:

- Если `x` классифицирована как переменная:
  - Вычисляется значение переменной `x`.
  - Вызывается выбранная операция со значением `x` в качестве аргумента.
  - Значение, возвращенное операцией, сохраняется в том же месте, где находилось значение `x`.
  - Значение, возвращенное операцией, становится результатом операции.
- Если `x` классифицируется как доступ к свойству или индексатору:
  - Вычисляются выражение экземпляра (если `x` не является `static`) и список аргументов (если `x` является доступом к индексатору), связанные с `x`, и результаты используются при последующих вызовах кодов доступа `get` и `set`.

- Вызывается код доступа `get` для `x`.
- Вызывается выбранная операция со значением аргумента, возвращенным кодом доступа `get`.
- Вызывается код доступа `set` для `x` со значением аргумента `value`, возвращенным операцией.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Если какой-либо из кодов доступа `get` или `set` отсутствует или недостижим, выдается ошибка компиляции.

- Значение, возвращенное операцией, становится результатом операции.

Операции `++` и `--` также поддерживают постфиксную нотацию (раздел 7.6.9). Обычно результатом `x++` или `x--` является значение `x` *перед* операцией, тогда как результатом `++x` или `--x` является значение *после* операции. В любом случае после выполнения любой из операций `x` будет иметь одно и то же значение.

> **ЭРИК ЛИППЕРТ**
>
> Стоит отметить, что в этом определении слово «обычно» ничего конкретного не означает. Действительно, так происходит в большинстве случаев, но нет такого *требования*, чтобы это было именно так. Значение `x` после операции может быть каким-нибудь старым значением; кто-нибудь может изменить значение `x` в другом потоке, `x` может быть свойством, которое возвращает среднее значение, когда вы вызываете код доступа `get` и т. д.
>
> Рассмотрим такой код:
>
> ```
> index = 0; value = this.arr[index++];
> ```
>
> Я часто слышу, как этот код описывают следующим образом: «Получаем элемент с индексом 0 и затем индекс увеличивается на единицу; прибавление происходит после поиска в массиве, поскольку `++` следует после `index`». Если вы прочитаете определение внимательно, вы увидите, что это утверждение совершенно неверно. В действительности порядок событий будет следующий: (1) получение текущего значения индекса; (2)»приращение индекса; и (3) поиск в массиве с использованием сохраненного значения. Приращение происходит *до* поиска, а не *после*.

Реализации `operator++` или `operator--` могут быть вызваны как с использованием постфиксной, так и префиксной нотации. Невозможно иметь две отдельные реализации операции для двух нотаций.

7.7.6. Выражения приведения

Выражения-приведения используются для явного приведения выражения к заданному типу.

```
выражение-приведения:
   ( тип )   унарное-выражение
```

Выражение-приведения в форме `(T)E`, где `T` является типом, а `E` является унарным выражением, выполняет явное приведение (раздел 6.2) значения `E` к типу `T`. Если не существует явного преобразования из `E` к `T`, выдается ошибка времени связывания. В противном случае результатом будет значение, полученное путем явного преобразования. Результат всегда классифицируется как значение, даже если `E` обозначает переменную.

ЭРИК ЛИППЕРТ

Операция приведения немного не похожа на другие как из-за необычного синтаксиса, так и из-за ее семантики. Операция приведения обычно используется, чтобы обозначить либо (1) я объявляю, что во время выполнения программы значение *не будет* иметь указанный в операции тип, и нужно что-нибудь предпринять, чтобы создать новое значение требуемого типа; либо (2) я объявляю, что во время выполнения программы значение *будет* иметь указанный в операции тип, и требуется это проверить и сгенерировать исключение, если я не прав. Внимательный читатель заметит, что это *противоположности*. Хотя и очевидно, что это какая-то хитрость — одна операция, производящая противоположные действия, но в результате легко сбиться с толку относительно того, что же происходит *на самом деле* при выполнении операции приведения.

Грамматика *выражения-приведения* ведет к некоторой синтаксической неоднозначности. Например, выражение `(x)–y` может быть интерпретировано либо как *выражение-приведения* (приведение `–y` к типу `x`), либо как *выражение-сложения* в сочетании с *выражением-в-скобках* (которое вычисляет значение `x – y`).

Чтобы разрешить неоднозначность, существует следующее правило: последовательность из одной и более *лексемы* (раздел 2.3.3), заключенная в скобки, рассматривается как начало *выражения-приведения* только в том случае, если выполняется как минимум одно из следующих условий:

- Последовательность символов представляет собой грамматику, корректную для *типа*, а не *выражения*.
- Последовательность символов представляет собой грамматику, корректную для *типа*, и лексема, непосредственно следующая за закрывающей скобкой, является «~», «`!`», «`(`», *идентификатором* (раздел 2.4.1), *литералом* (раздел 2.4.4) или любым *ключевым-словом* (2.4.3), кроме `as` и `is`.

Термин «корректная грамматика» здесь означает только то, что последовательность символов должна соответствовать определенному грамматическому правилу. Действительное значение любых составляющих идентификаторов не рассматривается. Например, если `x` и `y` — идентификаторы, то `x.y` является грамматикой, соответствующей типу, даже если `x.y` в действительности не обозначает тип.

Из этого правила для разрешения неоднозначности следует, что если `x` и `y` — идентификаторы, то `(x)y`, `(x)(y)` и `(x)(-y)` являются выражениями приведения, а `(x)-y` таковым не является, даже если `x` определяет тип. Однако если `x` является ключевым словом, которое определяет предопределенный тип (например, `int`), то все четыре формы являются выражениями приведения (поскольку такое ключевое слово само по себе не может быть выражением).

7.8. Арифметические операции

Операции *, /, %, +, и – называются арифметическими операциями.

```
выражение-умножения:
   унарное-выражение
   выражение-умножения   *   унарное-выражение
   выражение-умножения   /   унарное-выражение
   выражение-умножения   %   унарное-выражение

   выражение-сложения:
   выражение-умножения
   выражение-сложения    +   выражение-умножения
   выражение-сложения    -   выражение-умножения
```

Если операнд арифметической операции имеет тип времени компиляции `dynamic`, то выражение является динамически связанным (раздел 7.2.2). В этом случае тип времени компиляции выражения есть тип `dynamic`, и во время выполнения программы выполняется описанное ниже разрешение для типа тех операндов, которые имеют тип времени компиляции `dynamic`.

ПИТЕР СЕСТОФТ

Когда значение простого типа, например `double`, связано с переменной типа `dynamic`, во время выполнения программы оно обычно должно быть упаковано (храниться как объект в куче). Таким образом, арифметические вычисления с операндами типа `dynamic`, вероятно, производятся медленнее, чем вычисления с типами времени компиляции. Например, этот цикл выполняется приблизительно в шесть раз медленнее (на платформе Microsoft .NET 4.0), чем если бы `sum` была объявлена как переменная типа `double`:

```
dynamic sum = 0;
for (int i=0; i<count; i++)
sum += (i + 1.0) * i;
```

На самом деле это очень быстро по сравнению с арифметическими операциями в некоторых других языках с динамическими типами.

7.8.1. Операция умножения

Для операции в форме `x * y` разрешение перегрузки бинарной операции (раздел»7.3.4) применяется для выбора определенной реализации операции. Операнды приводятся к типам параметров выбранной операции, и типом результата будет тип, возвращаемый операцией.

Предопределенные операции умножения приведены ниже. Все эти операции вычисляют произведение `x` и `y`.

- Умножение целочисленных типов

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

В проверяемом контексте, если произведение выходит за границы значений типа результата, выбрасывается исключение `OverflowException`. В непроверяемом контексте о переполнении не сообщается, и старшие значащие биты, выходящие за границы значений типа результата, отбрасываются.

Умножение типов с плавающей точкой:

```
float operator *(float x, float y);
double operator *(double x, double y);
```

Произведение вычисляется в соответствии с правилами арифметики стандарта IEEE 754. В следующей таблице приведены результаты всех возможных комбинаций ненулевых конечных значений, нулевых значений, бесконечных значений и»NaN. В этой таблице `x` и `y` являются положительными конечными значениями; `z` является результатом умножения `x` и `y`. Если результат слишком велик для того, чтобы быть представленным в типе назначения, `z` считается бесконечным. Если результат слишком мал, `z` считается нулем.

	+y	–y	+0	–0	+∞	–∞	NaN
+x	+z	–z	+0	–0	+∞	–∞	NaN
–x	–z	+z	–0	+0	–∞	+∞	NaN
+0	+0	–0	+0	–0	NaN	NaN	NaN
–0	–0	+0	–0	+0	NaN	NaN	NaN
+∞	+∞	–∞	NaN	NaN	+∞	–∞	NaN
–∞	–∞	+∞	NaN	NaN	–∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Умножение для десятичного типа:

```
decimal operator *(decimal x, decimal y);
```

Если значение, получающееся в результате, слишком велико, чтобы быть представленным в формате `decimal`, выбрасывается исключение `System.OverflowException`. Если результирующее значение слишком мало, результат считается нулем. Масштабом результата перед округлением является сумма масштабов двух операндов.

Умножение для десятичного типа эквивалентно операции умножения для типа `System.Decimal`.

ПИТЕР СЕСТОФТ

Когда в тексте упоминается о «стандарте IEEE 754 для значений с плавающей точкой» — здесь и вообще везде в этой главе, было бы более точно говорить о «стандарте IEEE 754 для двоичных значений с плавающей точкой», поскольку версия 2008 этого стандарта IEEE описывает как двоичные значения с плавающей точкой, так и десятичные, и это очень разные вещи!

Также стандартом IEEE 754 установлено следующее правило для двоичных значений с плавающей запятой, соответствующее ныне использующимся реализациям Microsoft и Mono C#: если один или несколько операндов представляют собой NaN, результатом является NaN, и *данные, возвращаемые в результате, эквивалентны данным одного из*

операндов NaN. Данные NaN в 64-разрядном формате double содержат 51 младший разряд (так что тип double может представлять 251 — грубо говоря, 2*101 — различных значений NaN). Данные NaN в 32-разрядном формате float содержат 22 младших разряда (так что float может представлять 222 — грубо говоря, 4*106 — различных NaN). Сохранение данных NaN играет важную роль для эффективности научных расчетов. Для того чтобы такой подход реально работал, кроме арифметических операций также и математические функции должны работать с двоичными значениями IEEE с плавающей точкой. В реализациях, использующихся в настоящее время, такая возможность по большей части присутствует.
Библиотека .NET предоставляет метод System.Double.IsNan(d), для того чтобы определить, является ли NaN значением double.
Для типа double она также предоставляет методы DoubleToInt64Bits и Int64BitsToDouble, которые могут использоваться для получения и установки битов значений NaN:

```
public static long GetNanPayload(double d) {
   return System.BitConverter.DoubleToInt64Bits(d) & 0x0007FFFFFFFFFFFF;
}
public static double MakeNanPayload(long nanbits) {
   nanbits &= 0x0007FFFFFFFFFFFF;
   nanbits |= System.BitConverter.DoubleToInt64Bits(Double.NaN);
   return System.BitConverter.Int64BitsToDouble(nanbits);
}
```

Странно, что библиотека .NET не содержит соответствующих методов для типа float, но можно использовать небезопасные преобразования указателей для того, чтобы достичь такого же эффекта. Более подробно см. аннотацию раздела 18.4.

7.8.2. Операция деления

Для операции в форме x / y разрешение перегрузки применяется для выбора определенной реализации операции. Операнды приводятся к типам параметров выбранной операции, и типом результата является тип, возвращаемый операцией.

Предопределенные операции деления представлены ниже. Все эти операции вычисляют частное от деления x на y.

- Деление для целочисленных типов:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

Если значение правого оператора равно нулю, генерируется исключение System.DivideByZeroException.

При делении результат округляется в сторону нуля. Таким образом, абсолютное значение результата есть наибольшее возможное целое, которое меньше или равно абсолютному значению частного двух операндов. Результат будет нулем или положительным числом, когда два операнда имеют один и тот же знак, и будет нулем или отрицательным числом, когда два операнда имеют разные знаки.

Если левый операнд является наименьшим значением, которое может быть представлено типами `int` или `long`, а правый операнд имеет значение `–1`, возникает переполнение. В проверяемом контексте это приведет к появлению исключения `System.ArithmeticException` (или его подкласса). В непроверяемом контексте все определяется реализацией — будет ли сгенерировано исключение `System.ArithmeticException`, или же о переполнении не будет сообщено, и результатом станет значение левого операнда.

- Деление для типов с плавающей точкой:

```
float operator /(float x, float y);
double operator /(double x, double y);
```

Частное вычисляется в соответствии с правилами арифметики стандарта IEEE 754. В следующей таблице представлены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечностей и NaN. В данной таблице `x` и `y` являются положительными конечными значениями; `z` является результатом деления `x` на `y`. Если результат слишком велик для того, чтобы быть представленным в типе назначения, `z` считается бесконечным. Если результат слишком мал для того, чтобы быть представленным в типе назначения, `z` считается равным нулю.

	+y	–y	+0	–0	+∞	–∞	NaN
+x	+z	–z	+∞	–∞	+0	–0	NaN
–x	–z	+z	–∞	+∞	–0	+0	NaN
+0	+0	–0	NaN	NaN	+0	–0	NaN
–0	–0	+0	NaN	NaN	–0	+0	NaN
+∞	+∞	–∞	+∞	–∞	NaN	NaN	NaN
–∞	–∞	+∞	–∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Деление для десятичного типа:

```
decimal operator /(decimal x, decimal y);
```

Если значение правого операнда равно нулю, выбрасывается исключение `System.DivideByZeroException`. Если результирующее значение слишлом велико, чтобы быть представленным в формате `decimal`, выбрасывается исключение `System.OverflowException`. Если результирующее знавчение слишком мало, чтобы быть представленным в формате `decimal`, результат считается равным нулю. Масштабом результата является наименьший масштаб, который сохраняет результирующее значение, равное значению десятичного типа, ближайшему к точному математическому результату.

Деление для десятичных типов эквивалентно использованию операции деления для типа `System.Decimal`.

7.8.3. Операция взятия остатка от деления

Для операции в форме `x % y` разрешение перегрузки для бинарного оператора (раздел 7.3.4) применимо для выбора определенной реализации операции. Операнды

преобразуются к типам параметров выбранной операции, и типом результата будет тип, возвращаемый операцией.

Предопределенные операции остатка от деления представлены ниже. Все они вычисляют остаток от деления `x` на `y`.

- Операция остатка для целочисленных типов:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

Результатом операции `x % y` будет значение, полученное путем вычисления `x – (x / y) * y`. Если `y` равно нулю, генерируется исключение `DivideByZeroException`.

ЭРИК ЛИППЕРТ

Рассмотрим такой код:

```
static bool IsOdd(int x) { return x%2 == 1; }
```

Правильно ли это? Нет! Согласно сказанному выше, это то же самое, что

```
static bool IsOdd(int x) { return x-(x/2)*2 == 1; },
```

что неверно при x, равном -1. Правильный код будет следующий:

```
static bool IsOdd(int x) { return x%2 != 0; }
```

Если левый операнд представляет собой наименьшее значение типа `int` или `long` и правый операнд равен –1, генерируется исключение `System.OverflowException`. Исключение в результате операции `x % y` никогда не может появиться, если не выбрасывается исключение в результате операции `x / y`.

- Операция остатка для типов с плавающей точкой:

```
float operator %(float x, float y);
double operator %(double x, double y);
```

В следующей таблице представлены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечных значений и NaN. В этой таблице `x` и `y` являются положительными конечными значениями; `z` является результатом операции `x % y` и вычисляется как `x – n * y`, где `n` — наибольшее возможное целочисленное значение, меньшее или равное `x / y`. Такой метод вычисления остатка аналогичен методу, используемому для целочисленных операндов, но отличается от определения IEEE 754 (согласно которому `n` есть целочисленное значение, ближайшее к `x / y`).

	+y	–y	+0	–0	+∞	–∞	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
–x	–z	–z	NaN	NaN	–x	–x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
–0	–0	–0	NaN	NaN	–0	–0	NaN
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
–∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Операция остатка для десятичного типа:

```
decimal operator %(decimal x, decimal y);
```

Если значение правого операнда рано нулю, выбрасывается исключение `System.DivideByZeroException`. Масштаб результата перед любым округлением равен большему из масштабов двух операндов, а знак результата, если он ненулевой, такой же, как знак `x`.

Операция остатка для десятичного типа эквивалентна операции остатка для типа `System.Decimal` .

> **КРИС СЕЛЛЗ**
>
> Операция «остатка», известная также в других языках программирования как операция «по модулю» (modulo), иногда обозначается как «mod». Если вы реальный гик, то можете заметить, что широко используете слово modulo в нормальных человеческих выражениях вместо фразы «за исключением», например «Modulo testing, debugging, and documentation, we're ready to ship!»[1].

7.8.4. Операция сложения

Для операции в форме `x + y` разрешение перегрузки бинарной операции (раздел 7.3.4) применяется для выбора определенной реализации операции. Операнды приводятся к параметрам-типам выбранной операции, и тип результата будет типом, возвращаемым операцией.

Предопределенные операции сложения представлены ниже. Для арифметических типов и перечислений эти операции вычисляют сумму двух операндов. Когда один или оба операнда имеют тип `string`, предопределенные операции сложения выполняют объединение строковых представлений операндов.

- Сложение для целочисленных типов:

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

В проверяемом контексте, если сумма выходит за границы интервала значений результирующего типа, выбрасывается исключение `System.OverflowException`. В непроверяемом контексте о переполнении не сообщается и все старшие значащие биты, выходящие за пределы представления, отбрасываются.

- Сложение для типов с плавающей точкой:

```
float operator +(float x, float y);
double operator +(double x, double y);
```

Сумма вычисляется в соответствии с правилами арифметики стандарта IEEE 754. В следующей таблице представлены результаты всех возможных комбинаций

[1] «Все готово, за исключением тестирования, отладки и документации». — *Примеч. перев.*

ненулевых конечных значений, нулей, бесконечных значений и NaN. В данной таблице x и y — ненулевые конечные значения; z является результатом операции x + y. Если x и y имеют одинаковую величину, но противоположные знаки, z будет положительным нулем. Если значение x + y слишком велико для того, чтобы быть представленным в типе назначения, z будет бесконечностью с таким же знаком, как x + y.

	y	+0	–0	+∞	–∞	NaN
x	z	x	x	+∞	–∞	NaN
+0	y	+0	+0	+∞	–∞	NaN
–0	y	+0	–0	+∞	–∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
–∞	–∞	–∞	–∞	NaN	–∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Операция сложения для десятичного типа:

```
decimal operator +(decimal x, decimal y);
```

Если результирующее значение слишком велико, чтобы его можно было представить в формате decimal, выбрасывается исключение System.OverflowException. Масштаб результата перед любым округлением равен большему из масштабов двух операндов.

Сложение для десятичного типа эквивалентно сложению для типа System.Decimal.

- Сложение для перечислимого типа. Каждый перечислимый тип неявно осуществляет следующие предопределенные операции, где E является перечислимым типом и U является базовым типом для E:

```
E operator +(E x, U y);
E operator +(U x, E y);
```

Во время выполнения программы эти операции вычисляются в точности как (E)((U)x + (U)y).

> **БИЛЛ ВАГНЕР**
> Снова напоминаю, что сложение для перечислений может не дать результат в виде допустимого элемента перечисления. То же самое верно для всех арифметических операций.

- Объединение строк

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

эти перегрузки бинарной операции + выполняют объединение строк. Если один из операндов объединения строк есть null, подставляется пустая строка. Иначе любой

аргумент, не являющийся строкой, приводится к представлению в виде строки с помощью вызова виртуального метода ToString, унаследованного от типа object. Если ToString возвращает null, подставляется пустая строка.

```
using System;

class Test
{
   static void Main() {
      string s = null;
      Console.WriteLine("s = >" + s + "<");   // Выводится s = ><
      int i = 1;
      Console.WriteLine("i = " + i);          // Выводится i = 1
      float f = 1.2300E+15F;
      Console.WriteLine("f = " + f);          // Выводится f = 1.23E+15
      decimal d = 2.900m;
      Console.WriteLine("d = " + d);          // Выводится d = 2.900
   }
}
```

Результатом операции объединения строк является строка, состоящая из символов левого операнда, за которыми следуют символы правого операнда. Эта операция никогда не возвращает значение null. Если памяти для размещения результирующей строки недостаточно, может появиться исключение System.OutOfMemoryException.

> **ДЖОН СКИТ**
>
> Немного удивительно, что тип .NET System.String не имеет перегрузки для операции +, которая могла бы пригодиться в данной ситуации, пока вы не подумаете о возможностях оптимизации компилятора, связанных с видимостью большего фрагмента кода. Например, выражение a+b+c+d не должно давать при вычислении трех промежуточных строк: единственный вызов String.Concat(a, b, c, d) позволяет выполнить объединение более эффективно.

- Комбинация делегатов. Каждый тип делегата неявно осуществляет следующую предопределенную операцию, где D является типом делегата:

```
D operator +(D x, D y);
```

Бинарная операция + выполняет комбинацию делегатов, если оба операнда имеют один и тот же тип делегата D (если операнды имеют различные типы делегата, выдается ошибка времени связывания). Если первый операнд имеет значение null, результатом операции будет значение второго операнда (даже если он тоже имеет значение null). Если же значение второго операнда есть null, то результатом операции будет значение первого операнда. Иначе результатом операции будет новый экземпляр делегата, который, будучи вызван, вызывает первый операнд и затем вызывает второй операнд. Примеры комбинации делегатов можно найти в разделе 7.8.5 и в разделе 15.4. Так как System.Delegate не является типом делегата, операция + для него не определена.

7.8.5. Операция вычитания

Для операции в форме `x – y` разрешение перегрузки бинарной операции (раздел 7.3.4) применяется для выбора определенной реализации операции. Операнды приводятся к типам-параметрам выбранной операции, и типом результата будет тип, возвращаемый операцией.

Предопределенные операции вычитания представлены ниже. Все эти операции вычитают `y` из `x`.

- Вычитание для целочисленных типов:

```
int operator –(int x, int y);
uint operator –(uint x, uint y);
long operator –(long x, long y);
ulong operator –(ulong x, ulong y);
```

В проверяемом контексте, если разность выходит за границы интервала значений результирующего типа, выбрасывается исключение `System.OverflowException`. В непроверяемом контексте о переполнении не сообщается, и все старшие значащие биты, которые не помещаются в результирующий тип, отбрасываются.

- Вычитание для типов с плавающей точкой:

```
float operator –(float x, float y);
double operator –(double x, double y);
```

Разность вычисляется в соответствии с правилами арифметики стандарта IEEE. В следующей таблице представлены результаты всех возможных комбинаций ненулевых конечных значений, нулей, бесконечных значений и NaN. В этой таблице `x` и `y` — ненулевые конечные значения; `z` является результатом операции `x – y`. Если `x` и `y` равны, `z` будет положительным нулем. Если значение `x – y` слишком велико, чтобы быть представленным в типе назначения, `z` будет бесконечностью того же знака, как у `x – y`.

	y	+0	–0	+∞	–∞	NaN
x	z	x	x	–∞	+∞	NaN
+0	–y	+0	+0	–∞	+∞	NaN
–0	–y	–0	+0	–∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
–∞	–∞	–∞	–∞	–∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Вычитание для десятичного типа

```
decimal operator –(decimal x, decimal y);
```

Если результирующее значение слишком велико, чтобы быть представленным в формате `decimal`, генерируется исключение `System.OverflowException`. Масштаб результата перед любым округлением равен большему из масштабов двух операндов.

Вычитание для десятичного типа эквивалентно операции вычитания для типа `System.Decimal`.

- Вычитание для перечислимого типа. Каждый перечислимый тип неявно осуществляет следующую предопределенную операцию, где `E` является перечислимым типом и `U` является базовым типом для `E`:

```
U operator –(E x, E y);
```

Эта операция вычисляется в точности как `(U)((U)x – (U)y)`. Иными словами, операция вычисляет разность между перечисляемыми значениями `x` и `y`, и типом результата будет базовый тип данного перечислимого типа.

```
E operator –(E x, U y);
```

Эта операция вычисляется в точности как `(E)((U)x – y)`. Иными словами, происходит вычитание значения из базового типа перечисления, что в результате дает значение перечислимого типа.

- Удаление делегата. Каждый тип делегата неявно производит следующую предопределенную операцию, где `D` является типом делегата:

```
D operator –(D x, D y);
```

Бинарная операция – выполняет удаление делегата, когда оба операнда имеют один и тот же тип делегата `D`. Если операнды имеют различные типы делегата, выдается ошибка времени связывания. Если первый операнд имеет значение `null`, результатом операции будет `null`. Напротив, если второй операнд имеет значение `null`, то результатом операции будет значение первого операнда. Иначе оба операнда представляют списки вызовов (раздел 15.1), имеющие одну или более записей, и результатом будет новый список вызовов, состоящий из списка первого операнда за вычетом записей списка второго операнда, при условии что список второго операнда действительно является частью списка первого операнда. (Чтобы определить равенство частей списка, соответствующие записи сравниваются при помощи операции сравнения делегатов на равенство (раздел 7.10.8).) В противном случае результатом будет значение левого операнда. Ни один из списков операндов не изменяется в процессе выполнения операции. Если список второго операнда дублирует несколько частей списка первого операнда, состоящих из непрерывно следующих записей, наиболее правая повторяющаяся часть списка удаляется. Если в результате удаления получается пустой список, результатом будет `null`. Например:

```
delegate void D(int x);
class C
{
   public static void M1(int i) { /* ... */ }
public static void M2(int i) { /* ... */ }
}
class Test
{
   static void Main() {
      D cd1 = new D(C.M1);
      D cd2 = new D(C.M2);
      D cd3 = cd1 + cd2 + cd2 + cd1;   // M1 + M2 + M2 + M1
      cd3 -= cd1;                      // => M1 + M2 + M2

      cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
      cd3 -= cd1 + cd2;                // => M2 + M1
```

```
        cd3 = cd1 + cd2 + cd2 + cd1;    // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;               // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;    // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;               // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;    // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;               // => M1 + M2 + M2 + M1
    }
}
```

7.9. Операции сдвига

Операции << и >> используются для выполнения сдвига битов.

выражение-сдвига:
  выражение-сложения
  выражение-сдвига **<<** *выражение-сложения*
  выражение-сдвига *сдвиг-вправо* *выражение-сложения*

Если операнд выражения сдвига имеет тип времени компиляции `dynamic`, то выражение является динамически связанным (раздел 7.2.2). В этом случае тип выражения, определенный во время компиляции, есть `dynamic`, и разрешение, описанное ниже, будет происходить во время выполнения программы с использованием фактического типа операндов, имеющих тип времени компиляции `dynamic`.

Для операции в форме `x << count` или `x >> count` разрешение перегрузки (раздел 7.3.4) применяется для выбора определенной реализации операции. Операнды приводятся к параметрам-типам выбранной операции, и типом результата будет тип, возвращаемый операцией.

При объявлении перегруженной операции сдвига тип первого операнда всегда должен быть типом класса или структуры, содержащей объявление этой операции, а тип второго операнда всегда должен быть `int`.

Предопределенные операции сдвига представлены ниже.

- Сдвиг влево:

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

Операция `<<` сдвигает `x` влево на количество битов, которое вычисляется так, как описано ниже.

Старшие биты, выходящие за рамки интервала значений результирующего типа, отбрасываются, остающиеся биты сдвигаются влево, а пустые позиции младших битов устанавливаются равными нулю.

- Сдвиг вправо:

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

Операция `>>` сдвигает `x` вправо на количество битов, которое вычисляется так, как описано ниже.

Когда `x` имеет тип `uint` или `ulong`, младшие биты `x` отбрасываются, остающиеся биты сдвигаются вправо, и пустые позиции старших битов устанавливаются равными нулю.

Для предопределенных операций количество битов, на которое выполняется сдвиг, вычисляется следующим образом:

- Когда `x` имеет тип `int` или `uint`, величину сдвига дают пять младших битов `count`. Иными словами, величина сдвига вычисляется как `count & 0x1F`.
- Если `x` имеет тип `long` или `ulong`, величину сдвига дают шесть младших битов `count`. Иными словами, величина сдвига вычисляется как `count & 0x3F`.

Если в результате получается величина сдвига, равная нулю, операция сдвига просто возвращает значение `x`.

ДЖОН СКИТ

Маскирование величины сдвига может быть эффективным, но также может приводить к неожиданному поведению:

```
for (int i = 0; i < 40; i++)
{
   Console.WriteLine(int.MaxValue >> i);
}
```

Этот код будет печатать значения, которые на каждой итерации уменьшаются вдвое — до тех пор пока цикл не вернется назад к `int.MaxValue` после достижения нулевого значения.

Операции сдвига никогда не приводят к переполнению и дают один и тот же результат в проверяемом и непроверяемом контексте.

Когда левый операнд операции >> имеет целочисленный тип со знаком, операция выполняет *арифметический* сдвиг вправо, при котором значение самого старшего значащего бита (бита знака) операнда распространяется на пустые позиции старших битов. Когда левый операнд операции >> имеет целочисленный тип без знака, операция выполняет *логический* сдвиг вправо, при котором пустые позиции старших битов всегда устанавливаются равными нулю. Чтобы выполнить иную операцию, чем определено типом операнда, нужно выполнить явное приведение. Например, если `x` является переменной типа `int`, операция `unchecked((int)((uint)x >> y))` выполняет логический сдвиг `x` вправо.

7.10. Операции отношения и операции проверки типа

Операции `==`, `!=`, `<`, `>`, `<=`, `>=`, `is` и `as` называются операциями отношения и проверки типа.

```
выражение-отношения:
   выражение-сдвига
   выражение-отношения   <    выражение-сдвига
   выражение-отношения   >    выражение-сдвига
   выражение-отношения   <=   выражение-сдвига
   выражение-отношения   >=   выражение-сдвига
   выражение-отношения   is   тип
   выражение-отношения   as   тип

выражение-равенства:
   выражение-отношения
   выражение-равенства   ==   выражение-отношения
   выражение-равенства   !=   выражение-отношения
```

Операция `is` описана в разделе 7.10.10, операция `as` описана в разделе 7.10.11.

Операции `==`, `!=`, `<`, `>`, `<=` и `>=` являются *операциями-сравнения*.

Если операнд операции сравнения имеет тип времени компиляции `dynamic`, то выражение является динамически связанным (раздел 7.2.2). В этом случае тип времени компиляции выражения есть `dynamic`, и разрешение, описанное ниже, будет происходить во время выполнения программы с использованием фактического типа операндов, имеющих тип времени компиляции `dynamic`.

Для операции в форме `x` *op* `y`, где *op* является оператором сравнения, разрешение перегрузки (раздел 7.3.4) применяется для выбора определенной реализации операции. Операнды приводятся к параметрам-типам выбранной операции, и типом результата будет тип, возвращаемый операцией.

Предопределенные операции сравнения описаны в следующих разделах. Все предопределенные операции сравнения возвращают тип результата `bool`, как показано в следующей таблице.

Операция	**Результат**
`x == y`	`true`, если `x` равно `y`, `false` в противном случае
`x != y`	`true`, если `x` не равно `y`, `false` в противном случае
`x < y`	`true`, если `x` меньше чем `y`, `false` в противном случае
`x > y`	`true`, если `x` больше чем `y`, `false` в противном случае
`x <= y`	`true`, если `x` меньше или равно `y`, `false` в противном случае
`x >= y`	`true`, если `x` больше или равно `y`, `false` в противном случае

7.10.1. Операции сравнения для целочисленных типов

Предопределенные операции сравнения для целочисленных типов представлены ниже:

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);
```

продолжение ⇨

```
bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Каждая из этих операций производит сравнение численных значений двух целых операндов и возвращает значение типа `bool`, которое отражает, является ли данное отношение верным или неверным.

7.10.2. Операции сравнения для типов с плавающей точкой

Предопределенные операции сравнения для типов с плавающей точкой представлены ниже:

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

Эти операции производят сравнение операндов в соответствии с правилами стандарта IEEE 754:

- Если один из операндов является NaN, результат будет `false` для всех операций за исключением `!=`, для которой результат будет `true`. Для любых двух операндов `x != y` всегда дает такой же результат, как `!(x ==y)`. Однако, когда один или оба операнда являются NaN, операции `<`, `>`, `<=` и `>=` *не дают* того же результата, что логическое отрицание обратной операции. Например, если либо `x`, либо `y` есть NaN, то `x < y` дает результат `false`, но `!(x >= y)` дает `true`.
- Когда ни один из операндов не является NaN, операция сравнения величин двух операндов с плавающей точкой выполняется в следующем порядке:

  ```
  –∞ < –max < ... < –min < –0.0 == +0.0 < +min < ... < +max < +∞,
  ```

  где `min` и `max` являются соответственно наименьшим и наибольшим конечным положительным значением, которое может быть представлено в данном формате с плавающей точкой. Важные следствия этого порядка следующие:

  - Отрицательные и положительные нули считаются равными.
  - Отрицательная бесконечность считается меньшим значением, чем все другие значения, но равной другой отрицательной бесконечности.
  - Положительная бесконечность считается большим значением, чем все другие значения, но равной другой положительной бесконечности.

ПИТЕР СЕСТОФТ

Это также означает, что положительный и отрицательный ноль равны через `Object.Equals`, так что их хэш-коды, рассчитанные с помощью `Object.GetHashCode`, должны быть также равны. К сожалению, это так не во всех ныне существующих реализациях.

ДЖОЗЕФ АЛЬБАХАРИ

Два значения NaN, не будучи равными в соответствии с операцией ==, будут, однако, равными в соответствии с методом `Equals`:

```
double x = double.NaN;
Console.WriteLine(x == x);          // False
Console.WriteLine(x != x);          // True
Console.WriteLine(x.Equals(x));     // True
```

Вообще говоря, метод `Equals` для типов воплощает тот прицип, что объект должен быть равным самому себе. Без такого предположения списки и словари не могут действовать, поскольку не будет способа определить принадлежность элемента. Операции `==` и `!=`, однако, не подчиняются этому принципу.

ПИТЕР СЕСТОФТ

Даже NaNs, которые имеют различное внутреннее представление (см. аннотацию к разделу 7.8.1), равны через `Object.Equals`.

7.10.3. Операции сравнения для десятичного типа

Предопределенные операции сравнения для десятичного типа представлены ниже:

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Каждая из этих операций сравнивает численное значение двух операндов десятичного типа и возвращает значение типа `bool`, которое отражает, является данное отношение верным или неверным. Каждая операция сравнения для десятичного типа эквивалентна соответствующей операции сравнения или равенства для типа `System.Decimal`.

7.10.4. Операции равенства для булевского типа

Предопределенные операции сравнения для булевского типа представлены ниже:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

Результатом операции `==` будет значение `true`, если оба операнда `x` и `y` имеют значение `true` или если они оба имеют значение `false`. В противном случае результат имеет значение `false`.

Результатом операции `!=` будет значение `false`, если оба операнда `x` и `y` имеют значение `true` или если они оба имеют значение `false`. В противном случае результат имеет значение `true`. Когда операнды имеют тип `bool`, операция `!=` приводит к такому же результату, что и операция `^`.

7.10.5. Операции сравнения для перечислений

Каждый перечислимый тип неявно осуществляет следующие предопределенные операции сравнения:

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

Результат вычисления `x` *op* `y`, где `x` и `y` являются выражениями перечислимого типа `E`, который имеет базовый тип `U`, и *op* — одна из операций сравнения, будет в точности таким же, как результат вычисления `((U)x)` *op* `((U)y)`. Иными словами, операция сравнения для перечислимого типа просто производит сравнение целых значений базового типа двух операндов.

7.10.6. Операции равенства для ссылочных типов

Предопределенные операции равенства для ссылочных типов представлены ниже:

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

Эти операции возвращают результат сравнения двух ссылок в смысле определения их равенства или неравенства.

Поскольку предопределенные операции для ссылочного типа применимы к операндам типа `object`, они применимы ко всем типам, в которых не объявлены соответствующие элементы `operator ==` и `operator !=`. Напротив, любые применимые определенные пользователем операции равенства эффективно скрывают предопределенные операции равенства для ссылочного типа.

Для предопределенных операций равенства для ссылочного типа должно выполняться одно из следующих требований:

- Оба операнда являются либо значениями *ссылочного-типа*, либо литералом `null`. Более того, существует явное ссылочное преобразование (раздел 6.2.4) из типа каждого операнда к типу другого операнда.
- Один операнд является значением типа `T`, где `T` является параметром-типом, а другой операнд является литералом `null`. Более того, `T` не имеет ограничений для типа-значения.

Если ни одно из этих условий не выполняется, выдается ошибка времени связывания. Важные следствия этого правила следующие:

- Использование предопределенных операций для ссылочных типов для сравнения двух ссылок, которые были найдены различными во время связывания, приводит к ошибке времени связывания. Например, если типы операндов, определенные во время связывания, являются типами двух классов `A` и `B` и если ни `A`, ни `B` не наследуются один из другого, то эти два операнда не могут ссылаться на один и тот же объект. Таким образом, такая операция рассматривается как ошибка времени связывания.
- Предопределенные операции равенства для ссылочных типов никогда не производят операции упаковки для операндов типов-значений. Выполнять такие операции упаковки не имеет смысла, так как ссылки на вновь размещенные упакованные экземпляры неизбежно отличаются от всех других ссылок.
- Если операнд `T`, являющийся параметром-типом, сравнивается с `null` и его тип времени выполнения определяется как тип-значениие, результатом сравнения будет `false`.

В следующем примере выполняется проверка, является ли аргумент параметра-типа без ограничений константой `null`.

```
class C<T>
{
   void F(T x) {
      if (x == null) throw new ArgumentNullException();
      ...
   }
}
```

Конструкция `x == null` допустима, несмотря на то что `T` может представлять собой тип-значениие. В этом случае результатом просто будет `false`.

> **БИЛЛ ВАГНЕР**
>
> Если `C.F()` вызывается с `default(int?)`, выбрасывается исключение. Значение обнуляемого типа считается равным константе `null`, если `HasValue` есть `false`.

Для операции в форме `x == y` или `x != y`, если существует какая-либо применимая операция `operator ==` или `operator !=`, согласно правилам разрешения перегрузки операций (раздел 7.3.4) будет выбрана эта операция вместо преопределенной операции равенства для ссылочного типа. Однако всегда возможно выбрать предопределенную операцию равенства для ссылочного типа с помощью явного приведения одного или обоих операндов к типу `object`. Код в следующем примере:

```
using System;
class Test
{
   static void Main()
   {
      string s = "Test";
      string t = string.Copy(s);
      Console.WriteLine(s == t);
      Console.WriteLine((object)s == t);
      Console.WriteLine(s == (object)t);
      Console.WriteLine((object)s == (object)t);
   }
}
```

выполняет вывод

```
True
False
False
False
```

Переменные `s` и `t` ссылаются на два отдельных экземпляра `string`, содержащих одинаковые символы. Первое сравнение дает `True`, поскольку, когда оба операнда имеют тип `string`, выбирается предопределенная операция равенства для типа `string` (раздел 7.10.7). Все остальные сравнения дают `False`, поскольку, когда один или оба операнда имеют тип `object`, выбирается предопределенная операция равенства для ссылочного типа.

Отметим, что приведенная выше техника не имеет смысла для значимых типов.

Пример

```
class Test
{
   static void Main()
   {
      int i = 123;
      int j = 123;
      System.Console.WriteLine((object)i == (object)j);
   }
}
```

выведет `False`, поскольку в результате приведений создаются ссылки на два различных экземпляра упакованного значения `int`.

7.10.7. Операции равенства для строкового типа

Предопределенные операции равенства для строкового типа представлены ниже:

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Два значения типа `string` считаются равными, когда выполняется одно из следующих условий:

- Оба значения есть `null`.
- Оба значения представляют собой ненулевые ссылки на экземпляры строки, которые имеют одинаковую длину и одинаковые символы в каждой позиции.

Операции равенства для строкового типа сравнивают *значения* строк, а не *ссылки*. Когда два отдельных экземпляра строки содержат в точности одинаковые последовательности символов, значения строк равны, но ссылки различны. Как описано в разделе 7.10.6, для сравнения ссылок в строках вместо сравнения строковых значений могут быть использованы операции равенства для ссылочного типа.

7.10.8. Операции равенства для типов делегатов

Каждый тип делегата неявно осуществляет следующие предопределенные операции сравнения:

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Два делегата считаются равными при следующих условиях:

- Если какой-либо из делегатов есть `null`, они равны в том и только в том случае, если оба они имеют значение `null`.
- Если делегаты имеют различные типы времени выполнения, они никогда не могут быть равными.
- Если оба экземпляра делегата имеют список вызовов (раздел 15.1), эти экземпляры равны тогда и только тогда, когда их списки вызовов имеют одинаковую длину и каждая запись в одном списке вызова равна (в соответствии с правилами, приведенным ниже) соответствующей по порядку записи в другом списке вызовов.

Следующие правила определяют равенство записей списков вызовов:

- Если две записи списка вызовов ссылаются на один и тот же нестатический метод для одного и того же целевого объекта (как это определено операциями равенства для ссылок), то записи равны.
- Записи списков вызовов, произведенные при вычислении семантически идентичных *выражений-анонимной-функции* с одинаковым (возможно, пустым) набором «захваченных» внешних переменных, могут быть (но не обязательно) равными.

7.10.9. Операции равенства и константа null

Операции `==` и `!=` допускают, чтобы один операнд имел значение обнуляемого типа, а другой операнд был константой `null`, даже если не существует предопределенной или определенной пользователем операции (в расширенной или нерасширенной форме).

Для операции в одной из форм:

```
x == null     null == x      x != null     null != x,
```

где `x` является выражением обнуляемого типа, если при разрешении перегрузки (раздел 7.2.4) не найдено подходящей операции, результатом будет значение `x`, вычисленное с помощью свойства `HasValue`. Более точно, первые две формы преобразуются к `!x.HasValue`, а последние две формы преобразуются к `x.HasValue`.

7.10.10. Операция is

Операция `is` используется для динамической проверки соответствия фактического типа объекта данному типу. Результатом операции `E is T`, где `E` является выражением и `T` является типом, будет булевское значение, которое показывавет, может ли `E` быть успешно приведено к типу `T` при помощи преобразования ссылки, преобразования упаковки или преобразования распаковки. После того как аргументы-типы подставлены во все параметры-типы, данная операция выполняется следующим образом:

- Если `E` является анонимной функцией, выдается ошибка компиляции.
- Если `E` является группой методов или литералом `null`, если тип `E` является ссылочным или обнуляемым типом и значение `E` есть `null`, результатом операции будет `false`.
- Иначе, пусть `D` представляет динамический тип `E` следующим образом:
  - Если тип `E` является ссылочным типом, `D` является типом ссылки на экземпляр `E` времени выполнения.
  - Если тип `E` представляет собой обнуляемый тип, то `D` является базовым типом для этого обнуляемого типа.
  - Если тип `E` представляет собой необнуляемый тип-значениие, `D` является типом `E`.
- Результат операции зависит от `D` и `T` следующим образом:
  - Если `T` представляет собой ссылочный тип, результатом будет `true`, если `D` и `T` являются одним и тем же типом, если `D` является ссылочным типом и существует неявное ссылочное преобразование из `D` к `T`, а также если `D` является типом-значением и существует преобразование упаковки для `D` в `T`.
  - Если `T` представляет собой обнуляемый тип, результатом будет `true`, если `D` и `T` являются одним и тем же типом.
  - В противном случае результатом будет `false`.

Отметим, что операцией `is` определенные пользователем преобразования не рассматриваются.

7.10.11. Операция as

Операция `as` используется для явного приведения значения к данному ссылочному типу или обнуляемому типу. В отличие от выражения приведения (раздел 7.7.6), операция `as` никогда не приводит к исключению. Вместо этого, если указанное преобразование невозможно, результирующим значением будет `null`.

Для операции в форме `E as T`, `E` должно быть выражением и `T` должен быть ссылочным типом, параметр-тип должен иметь ссылочный или обнуляемый тип. Более того, должно выполняться по меньшей мере одно из следующих условий, или, в противном случае, будет выдана ошибка компиляции:

- Из `E` к `T` существует тождественное преобразование (раздел 6.1.1), неявное приведение к обнуляемому типу (раздел 6.2.3), неявное ссылочное приведение (раздел 6.1.6), преобразование упаковки (раздел 6.1.7), явное приведение к типам обнуляемому типу (раздел 6.2.3), явное ссылочное приведение (раздел 6.2.4) или преобразование распаковки (раздел 6.2.5).
- Тип `E` или `T` является открытым типом.
- `E` является литералом `null`.

Если тип `E` времени компиляции не является типом `dynamic`, операция `E as T` дает такой же результат, как

```
E is T ? (T)(E) : (T)null,
```

за исключением того, что `E` вычисляется только единожды. Можно ожидать, что компилятор оптимизирует выполнение `E as T` таким образом, чтобы выполнялась только одна проверка динамического типа, а не две, как предполагается в описанной выше расшифровке.

Если тип времени компиляции `E` есть `dynamic`, в отличие от операции приведения, операция `as` не является динамически связанной (раздел 7.2.2). Расшифровка в этом случае будет такой:

```
E is T ? (T)(object)(E) : (T)null
```

Отметим, что некоторые преобразования, например преобразования, определенные пользователем, невозможно использовать с операцией `as`, и вместо этого должны выполняться выражения приведения.

В примере

```
class X
{
   public string F(object o) {
      return o as string;
         //Верно: строка имеет ссылочный тип
   }
   public T G<T>(object o) where T: Attribute {
      return o as T;
         // Верно: T имеет ограничения класса
   }
   public U H<U>(object o) {
      return o as U;
         // ошибка: U не имеет ограничений
   }
}
```

известно, что параметр-тип `T` для `G` имеет ссылочный тип, поскольку он имеет ограничения класса. Параметр-тип `U` для `H`, однако, неизвестен; следовательно, использование операции `as` в `H` не разрешается.

КРИС СЕЛЛЗ

Если вы используете FxCop (а это должно быть так!), это не должно походить на:

```
if( x is Foo ) { ((Foo)x).DoFoo(); }
```

Вместо этого предпочтительнее сделать следующее:

```
Foo foo = x as Foo;
if( foo != null ) { foo.DoFoo(); }
```

Почему? Поскольку `is`, `as` и приведение, по существу, являются одной и той же базовой операцией .NET, лучше, чтобы операция выполнялась один раз, затем выполнялась проверка результата на `null`, и затем с ненулевым результатом проводились какие-либо действия, чем чтобы операция выполнялась дважды.

ДЖОЗЕФ АЛЬБАХАРИ

Хотя с технической точки зрения Крис прав, это микрооптимизация. Ссылочные преобразования мало затратны.

КРИС СЕЛЛЗ

Хотя Джозеф абсолютно прав, я больше говорил о безопасности при использовании FxCop, чем об оптимизации.

ДЖОЗЕФ АЛЬБАХАРИ

Некоторые предпочитают операцию `as` операции приведения. Преимущество операции `as` в том, что она делает очевидным, что преобразование не является арифметическим или определенным пользователем. Однако есть проблема с тем, чтобы во всех случаях отдавать предпочтение операции `as`, так как не всегда желательно, чтобы преобразование, не завершившееся успешно, давало в результате `null`. Для иллюстрации рассмотрим результат выполнения следующего кода в случае, когда объект, на который ссылается `s`, *не является* строкой:

```
int length1 = ((string) s).Length;   // Выбрасывается InvalidCastException
int length2 = (s as string).Length;  // Выбрасывается NullReferenceException
```

Первая строка дает несущее полезную информацию исключение `InvalidCastException`, тогда как вторая строка дает (неоднозначное) исключение `NullReferenceException`. (Является ли `s` константой `null` или имеет неверный тип?)

ДЖОН СКИТ

После операции `as` почти всегда используется тест на сравнение со значением `null` — тот, который наиболее соответствует типу оператора:

```
// Должен быть эквивалентен коду в примечании Криса
as if (Foo foo = x)
{
   foo.DoFoo();
}
```

Дополнительный оператор, может быть, и лиший, но неуклюжесть этого кода раздражает.

7.11. Логические операции

Операции `&`, `^` и `|` называются логическими операциями.

```
выражение-и:
   выражение-равенства
   выражение-и   &   выражение-равенства

выражение-исключающее-или:
   выражение-и
   выражение-исключающее-или   ^   выражение-и

выражение-включающее-или:
   выражение-исключающее-или
   выражение-включающее-или   |   выражение-исключающее-или
```

Если операнд логической операции имеет тип времени компиляции `dynamic`, то выражение является динамически связанным (раздел 7.2.2). В этом случае типом выражения времени компиляции будет тип `dynamic`, и разрешение, описанное ниже, будет происходить во время работы прогрммы с использованием фактического типа операндов, которые имели тип времени компиляции `dynamic`.

Для операции в форме `x` *op* `y`, где *op* является одной из логических операций, для выбора определенной реализации операции применяется разрешение перегрузки (раздел 7.3.4). Операнды преобразуются к параметрам-типам выбранной операции, и типом результата будет тип, возвращаемый операцией.

Предопределенные логические операции описаны в следующих разделах.

7.11.1. Логические операции для целочисленных типов

Предопределенные логические операции для целочисленных типов представлены ниже:

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);
```

продолжение ⇗

```
int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y)
```

Операция `&` вычисляет поразрядное логическое `AND` для двух операндов, операция | вычисляет поразрядное логическое `OR` для двух операндов, и операция ^ вычисляет поразрядное логическое исключающее `OR` для двух операндов. Переполнения при этих операциях невозможны.

7.11.2. Логические операции для перечислений

Любой перечислимый тип `E` неявно осуществляет следующие предопределенные логические операции:

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

Результат вычисления операции `x` *op* `y`, где `x` и `y` являются выражениями перечислимого типа `U` и *op* есть одна из логических операций, в точности такой же, как результат вычисления `(E)((U)x` *op* `(U)y)`. Иными словами, логические операции для перечислимого типа просто выполняют логические операции над базовым типом двух операндов.

7.11.3. Булевские логические операции

Предопределенные булевские логические операции представлены ниже:

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

Результатом операции `x & y` будет `true`, когда оба операнда `x` и `y` имеют значение `true`. В противном случае результатом будет `false`.

Результатом операции `x | y` будет `true`, когда либо `x`, либо `y` имеют значение `true`. В противном случае результатом будет `false`.

Результатом операции `x ^ y` будет `true`, когда либо `x` имеет значение `true`, а `y` имеет значение `false`, либо `x` имеет значение `false`, а `y` имеет значение `true`. Когда операнды имеют тип `bool`, операция `^` дает такой же результат, как операция `!=`.

7.11.4. Обнуляемые булевские логические операции

Обнуляемый булевский тип `bool?` может представлять три значения: `true`, `false` и `null`. Он напоминает трехзначный тип, используемый для булевских

выражений в SQL. Чтобы обеспечить согласование результатов операций `&` и `|` над операндами типа `bool?` с трехзначной логикой SQL, существуют следующие предопределенные операции:

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

В следующей таблице представлены результаты этих операций для всех комбинаций значений `true`, `false` и `null`.

x	**y**	**x & y**	**x | y**
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

7.12. Условные логические операции

Операции `&&` и `||` называются условными логическими операциями. Они также называются «короткими» логическими операциями.

условное-выражение-и
    выражение-включающее-или
    условное-выражение-и **&&** *выражение-включающее-или*

условное-выражение-или:
    условное-выражение-и
    условное-выражение-или **||** *условное-выражение-и*

Операции `&&` и `||` являются условными версиями операций `&` и `|`:

- Оперция `x && y` соответствует операции `x & y`, за исключением того, что `y` вычисляется, только если `x` не является `false`.
- Операция `x || y` соответствует операции `x | y`, за исключением того, что `y` вычисляется, только если `x` не является `true`.

Если операнд условной логической операции имеет тип времени компиляции `dynamic`, то выражение является динамически связанным (раздел 7.2.2). В этом случае типом выражения, определенным во время компиляции, будет тип `dynamic`, и разрешение, описанное ниже, будет происходить во время выполнения программы с использованием фактического типа тех операндов, которые имели тип времени компиляции `dynamic`.

Операция в форме `x && y` или `x || y` обрабатывается с применением разрешения перегрузки (раздел 7.3.4) так же, как операция, записанная в форме `x & y` или `x | y`. Тогда:

- Если с помощью разрешения перегрузки не удается найти единственную наилучшую операцию или если разрешение перегрузки выбирает одну из предопределенных логических операций для целочисленного типа, выдается ошибка времени связывания.
- Иначе, если выбранная операция является одной из предопределенных булевских логических операций (раздел 7.11.3) или обнуляемой булевской логической операцией (раздел 7.11.4), эта операция обрабатывается, как описано в разделе 7.12.1.
- Иначе выбранная операция является операцией, определенной пользователем, и она обрабатывается, как описано в разделе 7.12.2.

Невозможно перегрузить условные логические операции непосредственно. Однако поскольку вычисление условных логических операций дает те же значения, что и вычисление обычных логических операций, то перегрузки обычных логических операций могут рассматриваться с определенными ограничениями, так же как и перегрузки условных логических операций. Эта ситуация будет описана дальше в разделе 7.12.2.

7.12.1. Булевские условные логические операции

Когда операнды операций `&&` или `||` имеют тип `bool` или когда операнды имеют типы, в которых не определены подходящие операции `operator &` или `operator |`, но определены неявные преобразования к типу `bool`, операция обрабатывается следующим образом:

- Операция `x && y` вычисляется как `x ? y : false`. Иными словами, сначала вычисляется `x` и приводится к типу `bool`. Затем, если `x` имеет значение `true`, вычисляется `y` и приводится к типу `bool`, и это значение становится результатом операции. В противном случае результатом операции будет значение `false`.
- Операция `x || y` вычисляется как `x ? true : y`. Иными словами, сначала вычисляется `x` и приводится к типу `bool`. Затем, если `x` имеет значение `true`, результатом операции является `true`. В противном случае вычисляется `y` и приводится к типу `bool`, и это значение становится результатом операции.

7.12.2. Определенные пользователем условные логические операции

Когда операнды операций `&&` или `||` имеют типы, в которых объявлена подходящая определенная пользователем операция `operator&` или `operator |`, должны выполняться оба приведенных ниже условия. Здесь `T` является типом, в котором объявлена выбранная операция:

- Тип возвращаемого значения и тип каждого параметра выбранной операции должен быть `T`. Иными словами, эта операция должна вычислять логическое `AND` или логическое `OR` двух операндов типа `T` и должна возвращать результат типа `T`.
- `T` должен содержать объявления операций `operator true` и `operator false`.

Если хотя бы одно из этих требований не выполняется, выдается ошибка времени связывания. В противном случае операция `&&` или `||` вычисляется как комбинация определенной пользователем операции `operator true` или `operator false` и выбранной определенной пользователем операции:

- Операция `x && y` вычисляется как `T.false(x) ? x : T.&(x, y)`, где `T.false(x)` является вызовом операции `operator false`, объявленной в `T`, и `T.&(x, y)` является вызовом выбранной операции `operator &`. Иными словами, сначала вычисляется `x` и операция `operator false` вызывается, чтобы определить, точно ли `x` является `false`. Затем, если `x` действительно `false`, результатом операции становится значение, ранее вычисленное для `x`. В противном случае вычисляется `y` и выбранная операция `operator &` вызывается для значения, предварительно вычисленного для `x`, и для значения, вычисленного для `y`, чтобы получить результат операции.
- Операция `x || y` вычисляется как `T.true(x) ? x : T.|(x, y)`, где `T.true(x)` является вызовом операции `operator true`, объявленной в `T`, и `T.|(x, y)` является вызовом выбранной операции `operator |`. Иными словами, сначала вычисляется `x` и операция `operator true` вызывается, чтобы определить, точно ли `x` является `true`. Затем, если `x` действительно `true`, результатом операции становится значение, ранее вычисленное для `x`. В противном случае вычисляется `y` и выбранная операция `operator |` вызывается для значения, предварительно вычисленного для `x`, и для значения, вычисленного для `y`, чтобы получить результат операции.
- В каждой из этих операций выражение для `x` вычисляется только один раз, выражение для `y` вычисляется один раз или не вычисляется вовсе.

Пример типа, в котором реализованы `operator true` и `operator false`, можно найти в разделе 11.4.2.

7.13. Операция объединения с нулем

Операция `??` называется операцией объединения с нулем (null coalescing operator).

выражение-объединения-с-нулем:
  условное-выражение-или
  условное-выражение-или `??` *выражение-объединения-с-нулем*

Для выражения объединения с нулем в форме `a ?? b` требуется, чтобы `a` имела обнуляемый или ссылочный тип. Если `a` не является нулем, результатом `a ?? b` будет `a`; в противном случае результатом будет `b`. Вычисление `b` выполняется, только если `a` является `null`.

Операция объединения с нулем имеет правую ассоциативность, это означает, что действия группируются справа налево. Например, выражение в форме `a ?? b ?? c` вычисляется как `a ?? (b ?? c)`. В общем случае выражение в форме E_1 `??` E_2 `?? ...`

?? E_N возвращает первый из операндов, который является ненулевым, или null, если все операнды имеют значение null.

Тип выражения a ?? b зависит от того, какие неявные преобразования доступны для операндов. В порядке предпочтения, типом a ?? b будет A_0, A или B, где A является типом операнда a (при условии, что a имеет тип), B является типом b (при условии, что b имеет тип), и A_0 является базовым типом для A, если A есть обнуляемый тип, и является типом A в противном случае. Более точно, операция a ?? b обрабатывается следующим образом:

- Если A существует и не является обнуляемым или ссылочным типом, выдается ошибка компиляции.
- Если b является динамическим выражением, типом результата будет dynamic. Во время выполнения программы первым вычисляется a. Если a не является нулем, то a приводится к типу dynamic, и это значение становится результатом. В противном случае вычисляется b, и итог вычисления становится результатом.
- Иначе, если A существует и является обнуляемым типом и существует неявное преобразование из b к A_0, типом результата будет A_0. Во время выполнения программы первым вычисляется a. Если a не является нулем, a развертывается в тип A_0, и это значение становится результатом. В противном случае вычисляется b и преобразуется к типу A_0, и это значение становится результатом.
- Иначе, если A существует и существует неявное преобразование из b к A, типом результата будет A. Во время выполнения программы первым вычисляется a. Если a не является нулем, a становится результатом. В противном случае вычисляется b и преобразуется к типу A, и это значение становится результатом.
- Иначе, если b имеет тип B и существует неявное преобразование из a к B, типом результата будет B. Во время выполнения программы первым вычисляется a. Если a не является нулем, a развертывается в тип A_0 (если A существует и является обнуляемым типом) и приводится к типу B, и это значение становится результатом. В противном случае вычисляется b, и это значение становится результатом.
- Иначе a и b несовместимы, и выдается ошибка компиляции.

ЭРИК ЛИППЕРТ

Эти правила приведения значительно усложняют преобразования операции объединения с нулем в дерево выражений. В некоторых случаях компилятор может создать добавочное дерево выражений лямбда специально для того, чтобы управлять логикой преобразований.

КРИС СЕЛЛЗ

Операцию ?? удобно использовать для установки значений по умолчанию для ссылочных типов или для обнуляемых типов-значений.

Например:

```
Foo f1 = ...;
Foo f2 = f1 ?? new Foo(...);
int? i1 = ...;
int i2 = i1 ?? 452;
```

7.14. Условная операция

Операцию `?:` называют условной операцией, или иногда тернарной операцией.

условное-выражение:
  выражение-объединения-с-нулем:
  выражение-объединения-с нулем **?** *выражение* **:** *выражение*

Условное выражение в форме `b ? x : y` сначала вычисляет условие `b`. Затем, если `b` есть `true`, вычисляется `x`, и это значение становится результатом операции. В противном случае вычисляется `y`, и это значение становится результатом операции. Условное выражение никогда не вычисляет оба значения `x` и `y`.

Условная операция является правоассоциативной, это означает, что выполняемые действия группируются справа налево. Например, выражение в форме `a ? b : c ? d : e` вычисляется как `? b :(c ? d : e)`.

ДЖОН СКИТ

Я начал использовать составные условные операции случайно. Поначалу это смотрится странно, но код может быть весьма удобным для чтения, если они используются подходящим образом. В некотором смысле они приближают C# к операциям сравнения с образцом в таких языках, как F#.

```
return firstCondition ? firstValue :
   secondCondition ? secondValue :
   thirdCondition ? thirdValue :
   fallbackValue;
```

Первый операнд операции `?:` должен быть выражением, которое может быть неявно преобразуемо к типу `bool`, или выражением типа, в котором реализована операция `operator true`. Если ни одно из этих требований не выполняется, выдается ошибка компиляции.

ПИТЕР СЕСТОФТ

В частности, поскольку не существует неявного преобразования из обнуляемого типа `bool?` к `bool`, первый операнд условной операции не может иметь тип `bool?`.

Второй и третий операнды операции `?:`, `x` и `y`, определяют тип условного выражения.

- Если `x` имеет тип `X` и `y` имеет тип `Y`, тогда:
  - Если существует неявное преобразование (раздел 6.1) от `X` к `Y`, но не от `Y` к `X`, то типом условного выражения будет `Y`.
  - Если существует неявное преобразование (раздел 6.1) от `Y` к `X`, но не от `X` к `Y`, то типом условного выражения будет `X`.
  - Иначе тип выражения не может быть определен и выдается ошибка компиляции.
- Если только один из операндов `x` и `y` имеет тип, и оба они могут быть неявно преобразованы к этому типу, то этот тип будет типом условного выражения.
- Иначе тип выражения не может быть определен, и выдается ошибка компиляции.

ЭРИК ЛИППЕРТ

Компилятор Microsoft C# в действительности реализует немного отличающийся алгоритм: он проверяет преобразования из *выражений* к типам, а не из *типов* к типам. В большинстве случаев это различие не имеет значения, и изменение этого сейчас испортит существующий код.

ПИТЕР СЕСТОФТ

Несмотря на сказанное Эриком, существует небольшое нессответствие между компилятором Microsoft's C# 4.0 и предыдущими версиями: компилятор не может вывести тип правой части приведенного ниже условного выражения, мотивируя тем, что «не существует неявного преобразования между `‘<null>’` и `‘<null>’`»:

```
int? x = args.Length>0 ? null : null;
```

Компилятор C# 2.0 пропустил бы это выражение. Сообщение компилятора C# 4.0 намекает на возможное существование «нулевого типа» внутри компилятора или по меньшей мере в голове того, кто писал сообщение об ошибке. Учитывая, что это чрейзвычайно мудреный фрагмент кода, не стоит из-за этого терять сон.

ВЛАДИМИР РЕШЕТНИКОВ

Представление о «нулевом типе», существовавшее в предыдущих версиях спецификации C#, исчезло в C# 3.0, что привело к некоторому различию в поведении в сложных случаях. В настоящее время литерал `null` является выражением, которое не имеет типа.

Обработка условного выражения в форме `b ? x : y` состоит из следующих шагов:

- Во-первых, вычисляется `b` и определяется булевское значение для `b`:
  - Если существует неявное преобразование из типа `b` к типу `bool`, то выполняется это неявное преобразование, чтобы получить значение типа `bool`.
  - В противном случае вызывается операция `operator true`, определяемая типом `b`, для получения булевского значения.

- Если значение типа `bool`, полученное на предыдущем шаге, есть `true`, то вычисляется `x` и преобразуется к типу данного условного выражения, и это значение становится результатом условного выражения.
- В противном случае вычисляется `y` и преобразуется к типу данного условного выражения, и это значение становится результатом условного выражения.

> **БИЛЛ ВАГНЕР**
>
> Для меня операция объединения с нулем и условная операция подобны острым специям: в малых количествах они придают вкус. Крис указал на примеры, когда операция объединения с нулем приводит к созданию более ясного кода, чем при использовании эквивалентных операторов `if-then-else`. Подобно случаю с острыми специями, чрезмерное использование этих средств портит дело и делает не пригодным то, что могло быть весьма полезным.

7.15. Выражения анонимных функций

Анонимная-функция является выражением, которое представляет собой «встроенное» определение метода. Сама по себе анонимная функция не имеет значения или типа, но приводится к соответствующему типу делегата или дерева выражений. Преобразование, применяющееся к анонимной функции, зависит от целевого типа преобразования: если это тип делегата, в результате преобразования вычисляется значение делегата с помощью метода, определяемого анонимной функцией. Если это тип дерева выражений, в результате преобразования вычисляется дерево выражений, которое представляет структуру метода как структуру объекта.

По причинам исторического характера существует две разновидности анонимных функций — а именно, *лямбда-выражения* и *выражения-анонимных-методов*. Практически во всех случаях *лямбда-выражения* более кратки и выразительны, чем *выражения-анонимных-методов*, которые остаются в языке для обратной совместимости.

```
лямбда-выражение:
   сигнатура-анонимной-функции   =>   тело-анонимной-функции

выражение-анонимного-метода:
   delegate   сигнатура-явной-анонимной-функции_opt   блок

сигнатура-анонимной-функции:
   сигнатура-явной-анонимной-функции
   сигнатура-неявной-анонимной-функции

сигнатура-явной-анонимной-функции
   (   список-параметров-явной-анонимной-функции_opt   )

список-параметров-явной-анонимной-функции:
   параметр-явной-анонимной-функции
   список-параметров-явной-анонимной-функции   ,   параметр-явной-анонимной-функции
```

продолжение ☞

```
параметр-явной-анонимной-функции:
   модификатор-параметра-анонимной-функции opt   тип   идентификатор

модификатор-параметра-анонимной-функции:
   ref
   out

сигнатура-неявной-анонимной-функции:
   (  список-параметров-неявной-анонимной-функции opt  )
   параметр-неявной-анонимной-функции

список-параметров-неявной-анонимной-функции:
   параметр-неявной-анонимной-функции
   список-параметров-неявной-анонимной-функции  ,  параметр-неявной-анонимной-функции

параметр-неявной-анонимной-функции:
   иденитификатор

тело-анонимной-функции:
   выражение
   блок
```

Операция `=>` имеет такой же приоритет, как операция присваивания (`=`), и правую ассоциативность.

Парметры анонимной функции в форме *лямбда-выражения* могут иметь явно или неявно заданный тип. В списке параметров типа, заданного явным образом, тип каждого параметра явно установлен. В списке параметров типа, заданных неявно, типы параметров выводятся из контекста, в котором существует анонимная функция — точнее, когда анонимная функция приводится к совместимому типу делегата или к типу дерева выражений, этот тип определяет типы параметров (раздел 6.5).

> **БИЛЛ ВАГНЕР**
>
> В общем случае, анонимные функции более пластичны, если полагаться на неявное задание типов.

Для анонимной функции с единственным параметром с неявно заданным типом в списке параметров скобки могут быть опущены. Иными словами, форма записи анонимной функции

```
(   param   )   =>   expr
```

может быть сокращена до

```
param   =>   expr
```

Список параметров для функции в форме *выражения-анонимного-метода* необязателен. Если он задан, параметры должны быть заданы явно. Если он отсутствует, анонимная функция преобразуема к делегату с любым списком параметров, не содержащим параметров `out`.

Некоторые примеры анонимных функций представлены ниже:

```
x => x + 1                     // Неявное задание типов, тело — выражение
x => { return x + 1; }         // Неявное задание типов, тело — оператор
```

```
(int x) => x + 1                    // Явное задание типов, тело — выражение
(int x) => { return x + 1; }        // Явное задание типов, тело — оператор
(x, y) => x * y                     // Несколько параметров
() => Console.WriteLine()           // Нет параметров
delegate (int x) { return x + 1; }  // Выражение анонимного метода
delegate { return 1 + 1; }          // Список параметров опущен
```

Поведение *лямбда-выражений* и *выражений-анонимных-методов* одинаково за исключением следующих пунктов:

- *Выражения-анонимных-методов* позволяют полностью опускать список параметров, что означает преобразуемость к типам делегатов с любым списком параметров-значений.
- *Лямбда-выражения* позволяют полностью опускать или выводить список параметров, в то время как *выражения-анонимных-методов* требуют, чтобы типы параметров были заданы явно.
- Тело *лямбда-выражения* может быть выражением или блоком операторов, в то время как тело *выражения-анонимного-метода* должно быть блоком операторов.
- Так как только *лямбда-выражения* могут иметь тело в виде *выражения*, никакое *выражение-анонимного-метода* нельзя успешно преобразовать к типу дерева выражений (раздел 4.6).

> **БИЛЛ ВАГНЕР**
>
> Этот пункт важен для построения запросов, которые основаны на деревьях выражений, например, для используемых в Linq2SQL или в Entity Framework.

7.15.1. Сигнатуры анонимной функции

Необязательная *сигнатура-анонимной-функции* определяет имена и (необязательно) типы формальных параметров анонимной функции. Областью видимости параметров анонимной функции является *тело-анонимной-функции* (раздел 3.7). *Тело-анонимного-метода* вместе со списком параметров (если таковой существует) составляют область объявлений (раздел 3.3). Если имя параметра анонимной функции совпадает с именем локальной переменной, локальной константы или параметра, область видимости которых включает *выражение-анонимного-метода* или *лямбда-выражение*, выдается ошибка компиляции.

Если анонимная функция имеет *сигнатуру-явной-анонимной-функции*, то набор совместимых типов делегатов и деревьев выражений ограничивается теми типами, которые имеют те же типы параметров и модификаторы, указанные в том же порядке. В отличие от преобразований группы методов (раздел 6.6), контравариантность типов параметров анонимной функции не поддерживается. Если анонимная функция не имеет *сигнатуры-анонимной-функции*, то набор совместимых типов делегатов и деревьев выражений ограничивается теми типами, которые не имют параметров с модификатором `out`.

Отметим, что *сигнатура-анонимной-функции* не может включать в себя атрибуты и параметры-массивы. Тем не менее *сигнатура-анонимной-функции* может быть совместима с типом делегата, список параметров которого содержит параметр-массив.

Отметим также, что преобразование к дереву выражений, даже если оно является совместимым, может не пройти во время компиляции (раздел 4.6).

> **ЭРИК ЛИППЕРТ**
>
> Это тонкий момент. Если у вас есть две перегрузки — скажем, `void M(Expression<Func<Giraffe>> f)` и `void M(Func<Animal> f)`, и вызов `M(()=>myGiraffes[++i])`, то в качестве лучшей перегрузки выбирается перегрузка для дерева выражений. В этой ситуации выдается ошибка компиляции, поскольку операция инкремента недопустима внутри дерева выражений.

7.15.2. Тело анонимной функции

Тело анонимной функции (*выражение* или *блок*) подчиняется следующим правилам:

- Если анонимная функция содержит сигнатуру, параметры, определенные в сигнатуре, доступны внутри тела. Если анонимная функция не имеет сигнатуры, она может быть преобразована к типу делегата или к типу выражения с параметрами (раздел 6.5), но параметры в теле не будут доступны.
- За исключением случая, когда определены параметры `ref` или `out` в сигнатуре (если таковая имеется) непосредственно содержащей их анонимной функции, при попытке доступа из тела функции к параметрам `ref` или `out` выдается ошибка компиляции.
- Когда типом `this` является структурный тип, при попытке доступа из тела функции к `this` выдается ошибка компиляции. Это верно независимо от того, является ли доступ явным (как, например, в `this.x`) или неявным (как, например, в `x`, когда `x` является элементом экземпляра структуры). Это правило просто запрещает такой доступ, независимо от результатов поиска элемента структуры.
- Тело функции имеет доступ к внешним переменным (раздел 7.15.5) анонимной функции. Доступ к внешней переменной будет ссылаться на экземпляр переменной, который действует во время вычисления *лямбда-выражения* или *выражения-анонимного-метода* (раздел 7.15.6).
- Если тело функции содержит операторы `goto`, `break` или `continue`, переход из которых находится вне тела функции или внутри содержащейся в теле другой анонимной функции, выдается ошибка компиляции.
- Оператор `return` в теле функции возвращает управление из вызова непосредственно содержащей его анонимной функции, а не из объемлющего элемента-функции. Выражение, определенное в операторе `return`, должно быть неявно преобразуемо к типу возвращаемого значения типа делегата или дерева вы-

ражений, к которому преобразуется непосредственно содержащее его *лямбда-выражение* или *выражение-анонимного-метода* (раздел 6.5).

Явно не определено, существует ли какой-либо способ выполнить блок анонимной функции помимо вычисления и вызова *лямбда-выражения* или *выражения-анонимного-метода*. В частности, компилятор может реализовать анонимную функцию, синтезировав один или несколько именованных методов или типов. Имена таких синтезированных элементов должны быть зарезервированы для использования компилятором.

7.15.3. Разрешение перегрузки

Анонимные функции в списке аргументов принимают участие в выведении типа и разрешении перегрузки. Точные правила приведены в разделе 7.4.2.3.

Следующий пример иллюстрирует воздействие анонимных функций на разрешение перегрузки.

```
class ItemList<T>: List<T>
{
   public int Sum(Func<T,int> selector) {
      int sum = 0;
      foreach (T item in this) sum += selector(item);
      return sum;
   }
   public double Sum(Func<T,double> selector) {
      double sum = 0;
      foreach (T item in this) sum += selector(item);
      return sum;
   }
}
```

Класс `ItemList<T>` имеет два метода `Sum`. Каждый из них использует аргумент `selector`, который извлекает значение суммы из позиции списка. Извлеченное значение может иметь тип `int` или `double`, и результирующая сумма также будет типа `int` или `double`.

Методы `Sum` могут быть, например, использованы для вычисления сумм в каждой строке списка по порядку.

```
class Detail
{
   public int UnitCount;
   public double UnitPrice;
   ...
}
void ComputeSums() {
   ItemList<Detail> orderDetails = GetOrderDetails(...);
   int totalUnits = orderDetails.Sum(d => d.UnitCount);
   double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
   ...
}
```

При первом вызове `orderDetails.Sum` оба метода `Sum` применимы. Поскольку анонимная функция `d => d.UnitCount` совместима как с `Func<Detail,int>`, так

и с `Func<Detail,double>`. Однако при разрешении перегрузки выбирается первый метод `Sum`, поскольку приведение к `Func<Detail,int>` лучше, чем приведение к `Func<Detail,double>`.

При втором вызове `orderDetails.Sum` применим только второй метод `Sum`, поскольку анонимная функция `d => d.UnitPrice * d.UnitCount` дает значение типа `double`. Таким образом, при разрешении перегрузки для вызова выбирается второй метод `Sum`.

7.15.4. Анонимные функции и динамическое связывание

Анонимная функция не может быть получателем, аргументом или операндом динамически связанной операции.

7.15.5. Внешние переменные

Любая локальная переменная, параметр-значение или параметр-массив, область видимости которых включает в себя *лямбда-выражение* или *выражение-анонимного-метода*, называется **внешней переменной** анонимной функции. В экземпляре функционального элемента класса значение `this` рассмаривается как параметр-значение и является внешней переменной любой анонимной функции, содержащейся внутри этого функционального элемента.

> **БИЛЛ ВАГНЕР**
> Это формальное определение реализации замыкания в C#. Это большое достижение.

7.15.5.1. Захваченные внешние переменные

Когда анонимная функция ссылается на внешнюю переменную, говорят, что внешняя переменная **захвачена** анонимной функцией. Обычно время жизни локальной переменной ограничено выполнением блока или оператора, с которым она связана (раздел 5.1.7). Однако время жизни захваченной внешней переменной увеличивается по меньшей мере до того момента, как делегат или дерево выражений, созданные из анонимной функции, становятся подлежащими сборке мусора.

> **БИЛЛ ВАГНЕР**
> Заметим, что в следующем примере `x` имеет большее время жизни, чем можно было бы ожидать, поскольку она захвачена в результате выполнения анонимного метода. Если `x` является ценным ресурсом, такое поведение следует пресечь, ограничив время жизни анонимного метода.

В примере

```
using System;

delegate int D();

class Test
{
   static D F() {
      int x = 0;
      D result = () => ++x;
      return result;
   }
   static void Main() {
      D d = F();
      Console.WriteLine(d());
      Console.WriteLine(d());
      Console.WriteLine(d());
   }
}
```

локальная переменная `x` захвачена анонимной функцией, и время жизни `x` увеличивается по меньшей мере до того момента, как делегат, возвращенный от `F`, становится подлежещим сборке мусора (которая не произойдет до самого конца программы). Так как каждый вызов анонимной функции действует на тот же самый экземпляр `x`, вывод в этом примере будет

```
1
2
3
```

Когда локальная переменная или параметр-значение захвачены анонимной функцией, эта локальная переменная или параметр-значение не может больше рассматриваться как фиксированная переменная (раздел 18.3), напротив, они рассматриваются как «перемещаемые» (moveable) переменные. Таким образом, любой небезопасный код, который использует адрес захваченной внешней переменной, прежде всего, должен содержать оператор `fixed`, чтобы зафиксировать эту переменную.

7.15.5.2. Инстанцирование локальных переменных

Локальная переменная считается **инстанцированной**, когда при выполнении программы выполняется вход в область видимости переменной. Например, когда вызывается приведенный ниже метод, локальная переменная `x` инстанцируется и инициализируется три раза — по одному разу на каждой итерации цикла:

```
static void F() {
   for (int i = 0; i < 3; i++) {
      int x = i * 2 + 1;
      ...
   }
}
```

Однако если переместить объявление `x` за пределы цикла, она будет инстанцирована только один раз:

```
static void F() {
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      ...
   }
}
```

Когда локальная переменная не захвачена, не существует способа точно установить, сколько раз она была инстанцирована: поскольку времена жини отдельных инстанцирований не пересекаются, все инстанцирования могут просто использовать одну и ту же ячейку хранения. Однако когда анонимная функция захватывает локальную переменную, действие инстанцирования становится видимым.

Пример

```
using System;

delegate void D();

class Test
{
   static D[] F() {
      D[] result = new D[3];
      for (int i = 0; i < 3; i++) {
         int x = i * 2 + 1;
         result[i] = () => { Console.WriteLine(x); };
      }
      return result;
   }
   static void Main() {
      foreach (D d in F()) d();
   }
}
```

выполняет следующий вывод:

```
1
3
5
```

Однако когда объявление `x` переносится за границы цикла

```
static D[] F() {
   D[] result = new D[3];
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      result[i] = () => { Console.WriteLine(x); };
   }
   return result;
}
```

вывод будет

```
5
5
5
```

Когда в цикле `for` объявляется переменная цикла, такая переменная считается объявленной за пределами цикла. Таким образом, если изменить пример так, что сама переменная цикла окажется захваченной,

```
static D[] F() {
   D[] result = new D[3];
   for (int i = 0; i < 3; i++) {
      result[i] = () => { Console.WriteLine(i); };
   }
   return result;
}
```

то окажется захваченным только один экземпляр переменной цикла, который обеспечит следующий вывод:

```
3
3
3
```

ЭРИК ЛИППЕРТ

Такое поведение — когда анонимные функции захватывают переменные цикла, а не текущее значение переменой — является причиной наиболее часто встречающегося заблуждения. «Я думаю, что нашел ошибку в компиляторе», — примерно такое сообщение можно услышать в подобном случае. Для последующих версий этой спецификации мы рассматриваем возможность перемещения формального определения переменной цикла `foreach` внутрь цикла, так что анонимная функция внутри цикла будет захватывать новую переменную на каждой итерации цикла. Технически это означает значительные изменения, но это приведет язык к такому состоянию, когда для большинства людей его поведение будет соответствовать ожидаемому. Трудно придумать ситуацию, в которой вы *хотите* захватить переменную цикла как переменную, а не как значение.

Делегаты анонимной функции могут совместно использовать некоторые захваченные переменные и в то же время иметь различные экземпляры других переменных. Например, если `F` изменяется следующим образом:

```
D[] result = new D[3];
int x = 0;
for (int i = 0; i < 3; i++) {
   int y = 0;
   result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
}
return result;
}
```

то три делегата захватывают один и тот же экземпляр `x`, но различные экземпляры `y`, и вывод будет:

```
1 1
2 1
3 1
```

Различные анонимные функции могут захватывать один и тот же экземпляр внешней переменной. В примере

```
using System;
delegate void Setter(int value);
delegate int Getter();
class Test
{
   static void Main() {
      int x = 0;
      Setter s = (int value) => { x = value; };
      Getter g = () => { return x; };
      s(5);
      Console.WriteLine(g());
      s(10);
      Console.WriteLine(g());
   }
}
```

две анонимные функции захватывают один и тот же экземпляр локальной переменной `x`, и они могут «общаться» через этот экземпляр. Вывод в этом примере будет:

```
5
10
```

7.15.6. Вычисление выражений анонимных функций

Анонимная функция `F` всегда должна преобразовываться к типу делегата `D` или к типу дерева выражений `E` либо непосредственно, либо через выполнение выражения создания делегата `new D(F)`. Это преобразование определяет результат анонимной функции, как описано в разделе 6.5.

7.16. Выражения запроса

Выражения запроса обеспечивают интегрированный с языком синтаксис для запросов, подобный существующему в языках, использующих реляционные и иерархические запросы, таких как SQL и XQuery.

```
выражение-запроса:
   конструкция-from   тело-запроса

конструкция-from
   from   тип_opt   идентификатор   in   выражение

тело-запроса:
   конструкции-тела-запроса_opt   конструкция-select-или-group
продолжение-запроса_opt

конструкции-тела-запроса:
   конструкция-тела-запроса
   конструкции-тела-запроса   конструкция-тела-запроса
```

конструкция-тела-запроса:
  конструкция-from
  конструкция-let
  конструкция-where
  конструкция-join
  конструкция-join-into
  конструкция-orderby

конструкция-let
  let *идентификатор* **=** *выражение*

конструкция-where:
  where *булевское-выражение*

конструкция-join:
  join *тип*$_{opt}$ *идентификатор* **in** *выражение* **on** *выражение* **equals** *выражение*

конструкция-join-into
  join *тип*$_{opt}$ *идентификатор* **in** *выражение* **on** *выражение* **equals** *выражение*
  into *идентификатор*

конструкция-orderby:
  orderby *упорядочения*

упорядочения:
  упорядочения **,** *упорядочение*

упорядочение:
  выражение *порядок-сортировки*$_{opt}$

порядок-сортировки:
  ascending
  descending

конструкция-select-или-group:
  конструкция-select
  конструкция-group

конструкция-select:
  select *выражение*

конструкция-group:
  group *выражение* **by** *выражение*

продолжение-запроса:
  into *идентификатор* *тело-запроса*

Выражение запроса начинается с конструкции `from` и заканчивается либо конструкцией `select`, либо конструкцией `group`. За начальной конструкцией `from` могут следовать ноль или более конструкций `from`, `let`, `where`, `join` и `orderby`. Каждая конструкция `from` вводит **переменную диапазона** (*range variable*), которая принимает ряд значений элементов **последовательности**. Каждая конструкция `let` вводит переменную диапазона, представляющую значение, вычисленное с помощью предыдущих переменных диапазона. Каждая конструкция `where` является фильтром, который исключает некоторые элементы из результата. Каждая

конструкция `join` сравнивает определенные ключи исходной последовательности с ключами другой последовательности, отыскивая подходящие пары. Каждая конструкция `orderby` упорядочивает элементы в соответствии с определенными критериями. Заключительная конструкция `select` или `group` определяет форму результата на основе переменных диапазона. Наконец, конструкцию `into` можно использовать для «склеивания» запросов, когда результаты одного запроса генерируют следующий запрос.

ЭРИК ЛИППЕРТ

Почему используется синтаксис «`from`...`where`...`select`», тогда как порядок, более привычный разработчикам SQL, «`select`...`from`...`where`»? Хотя порядок, принятый в SQL, имеет то преимущество, что он более привычен для разработчиков SQL и естественен для англоговорящих пользователей, он создает множество проблем для создателей языка и интегрированной среды разработки (Integrated Development Environment, IDE), так что такой порядок в C# не используется.

Во-первых, с принятым в SQL порядком затруднительно использовать IntelliSense из-за его правил видимости. Представьте, что вы конструируете IDE с применением синтаксиса «сначала `select`». Пользователь только что напечатал «`select`»: что теперь? Вы не знаете, из какого источника пользователь переносит данные, так что вы не можете составить разумный список нужных параметров. Вы также не сможете обеспечить никакой поддержки для «`where`». В системе C# «`from`» идет раньше «`select`»; таким образом, к тому времени как пользователь напечатает «`where`» или «`select`», IDE имеет информацию об источнике данных.

Во-вторых, принятый в C# порядок — это порядок, при котором операции действительно выполняются в соответствии с кодом: сначала определяется коллекция, затем на коллекцию накладывается фильтр и отображается то, что получилось в результате действия этого фильтра. Такой порядок помогает осознать, каким образом потоки данных проходят через систему.

КРИС СЕЛЛЗ

Теперь, когда я могу сравнить порядок «`select-from`» в SQL и «`from-select`» в LINQ, LINQ для меня имеет гораздо больше смысла, и я испытываю затруднения от того, что SQL не допускает стиль «`from-select`».

7.16.1. Неоднозначность в выражениях запроса

Выражения запроса содержат ряд «ключевых слов, зависящих от контекста», то есть идентификаторов, которые имеют специальное значение в данном контексте. Такими ключевыми словами являются `from`, `where`, `join`, `on`, `equals`, `into`, `let`, `orderby`, `ascending`, `descending`, `select`, `group` и `by`. Чтобы предотвратить неоднозначность в выражениях запроса, которая может появиться из-за одновременного использования этих идентиикаторов в качестве ключевых слов и в качестве простых имен, эти идентификаторы всегда считаются ключевыми словами, когда они появляются внутри выражения запроса.

При этом выражением запроса считается любое выражение, которое начинается с «`from` *идентификатор*», следующего за любым символом, за исключением «`;`», «`=`» и «`,`».

Для использования этих слов как идентификаторов внутри выражения запроса они должны иметь префикс «`@`». (раздел 2.4.2).

7.16.2. Преобразование выражений запроса

Язык C# не конкретизирует семантику выполнения выражений запроса. Выражения запроса преобразуются в вызовы методов, которые соответствуют *паттерну выражений запроса* (раздел 7.16.3). Выражения запроса преобразуются в вызовы методов, именуемых `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy` и `Cast`. Эти методы должны иметь сигнатуры и типы результатов, описанные в разделе 7.16.3. Эти методы могут быть методами экземпляра запрашиваемого объекта или методами расширения, являющимися внешними по отношению к объекту, и они обеспечивают фактическое выполнение запроса.

Преобразование выражений запроса в вызовы методов является синтаксическим отображением, осуществляющимся перед выполнением любого связывания типов или разрешения перегрузки. Это преобразование гарантированно является синтаксически корректным, но не гарантировано, что оно производит семантически корректный код C#. Получившиеся в результате преобразования выражений запроса вызовы методов обрабатываются как обычные вызовы методов, и здесь, в свою очередь, могут обнаружиться ошибки — например, если методы не существуют, если аргументы имеют неверные типы или если методы являются обобщенными и не удается выполнить выведение типов.

БИЛЛ ВАГНЕР

Этот раздел дает хорошую возможность понять, каким образом выражения запроса преобразуются в вызовы методов или в вызовы методов расширения.

ДЖОН СКИТ

Тот факт, что выражения запроса могут быть введены в C# с таким небольшим добавлением к спецификации, не перестает удивлять меня. В то время как некоторые объекты, например обобщенные типы, оказывают влияние практически на каждую область языка, выражения запроса удивительным образом замкнуты, в особенности если учесть большие возможности, которые они предоставляют.

При обработке выражения запроса происходит повторяющееся применение последовательных преобразований до тех пор, пока возможно дальнейшее упрощение. Преобразования перечислены в порядке их применения: каждая секция предполагает, что преобразования предыдущей секции выполнены до конца; однажды

выполнив их до конца, процесс обработки того же самого выражения запроса не возвращается более к той же секции.

Присваивание значений переменным диапазона в выражениях запроса недопустимо. Однако реализация C# позволяет иногда обходить это ограничение, так как порой это невозможно для представленной здесь схемы синтаксического преобразования.

Некоторые преобразования вводят переменные диапазона с *прозрачными идентификаторами*, обозначаемыми *. Свойства прозрачных идентификаторов обсуждаются далее в разделе 7.16.2.7.

КРИС СЕЛЛЗ

Хотя мне и нравится синтаксис запросов в версии C# 3.0, иногда удержать в голове преобразования трудно. Не стоит расстраиваться, если вы иной раз чувствуете необходимость записать запрос с использованием синтаксиса вызова метода. Кроме того, некоторые методы запросов, которые вы реализуете самостоятельно, не имеют соответствующих языковых конструкций, так что в этих случаях у вас нет иного выбора, кроме как использовать синтаксис вызова метода.

Далее, будет вполне правильно, если вы напишете запросы, подобные следующему:

```
var duluthians = from c in Customers
   where c.City == "Duluth" select c;
```

или же напишете запросы таким образом:

```
var duluthians = Customers.Where(c => c.City == "Duluth");
```

Во втором случае синтаксис не требует умственных усилий по преобразованию и работает для всех методов расширения, а не только для преобразованных в ключевые слова. Например:

```
var tenDuluthites = Customers.Where(c => c.City == "Duluth").Take(10);
```

против

```
var tenDuluthites = (from c in Customers where c.City ==
            "Duluth" select c).Take(10);
```

Я, как человек простой, предпочитаю синтаксический стиль, который работает всегда, а не то, что нынче делают крутые парни.

ДЖОН СКИТ

Хотя не похоже, что добавочные методы запросов для `IEnumerable<T>` поддерживаются выражениями запросов, одним из чудесных аспектов их определения является их нейтральность: нигде в спецификации не указано, какими должны быть типы. Это позволяет писать методы запросов с использованием новых типов на других платформах (например, Reactive Extensions и Parallel Extensions), сохраняя преимущества синтаксиса выражения запроса.

7.16.2.1. Конструкции select и GroupBy с продолжениями

Выражение запроса с продолжением

```
from ... into x ...
```

преобразуется в

```
from x in ( from ... ) ...
```

Преобразования, описанные в следующих разделах, предполагают, что запросы не имеют продолжений `into`.

Пример

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

преобразуется в

```
from g in
   from c in customers
   group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

и окончательно в

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

ДЖОЗЕФ АЛЬБАХАРИ

Продолжение запроса обеспечивает возможность после конструкций `select` или `group` (которые в противном случае заканчивают запрос) добавлять дальнейшие конструкции. После продолжения запроса переменная диапазона и другие переменные, описанные в конструкциях `join` или `let`, оказываются вне области видимости. В противоположность этому, `let` действует подобно неразрушающей конструкции `select`: она сохраняет переменную диапазона и другие переменные запроса в области видимости.

Идентификатор, используемый в продолжении запроса, может быть тем же, как у предшествующей переменной диапазона.

7.16.2.2. Явно определенные типы переменных диапазона

Конструкция `from`, в которой явно определент тип переменной диапазона

```
from T x in e
```

преобразуется в

```
from x in ( e ) . Cast < T > ( )
```

Конструкция `join,` в которой явно определен тип переменной диапазона

```
join T x in e on k1 equals k2
```

преобразуется в

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

Преобразования, описанные в следующих разделах, предполагают, что запросы не используют явно определенных типов переменных диапазона.

Пример

```
from Customer c in customers
where c.City == "London"
select c
```

преобразуется в

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

и окончательно в

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Явное определение типов переменных диапазона полезно для коллекций запросов, которые реализуют необобщенный интерфейс `IEnumerable`, но не для тех, которые реализуют обобщенный интерфейс `IEnumerable<T>`. В примере, приведенном выше, это соответствует случаю, когда `customers` имел бы тип `ArrayList`.

7.16.2.3. Вырожденные выражения запросов

Выражение запроса в форме

```
from x in e select x
```

преобразуется в

```
( e ) . Select ( x => x )
```

Пример

```
from c in customers
select c
```

преобразуется в

```
customers.Select(c => c)
```

Вырожденное выражение запроса просто выбирает элементы из источника. Дальнейшая фаза преобразования удаляет вырожденные запросы, полученные на предыдущих шагах преобразования, заменяя их на их источник. Важно, однако, убедиться, что результатом выражения запроса никогда не может являться сам исходный объект, так как это открывало бы тип исходного объекта и сам объект клиенту запроса. Таким образом, этот шаг защищает вырожденные запросы, записанные непосредственно в исходном коде, путем явного вызова `Select` для исходного объекта. То, что `Select` и другие операции запроса никогда не возвращают сам исходный объект, является областью ответственности реализции этих операций.

7.16.2.4. Конструкции from, let, where, join и orderby

> **ДЖОЗЕФ АЛЬБАХАРИ**
>
> Кажущиеся громоздкими преобразования этого раздела представляют то, что делает синтаксис запросов действительно полезным: они отменяют необходимость писать громоздкие запросы вручную. Без решения этой проблемы было бы мало оправдания введению синтаксиса выражений запроса в C# 3.0; данные возможности обеспечивают лямбда-выражения и методы расширения.
>
> Общей идеей более сложных преобразований в процессе отображения во временный анонимный тип является сохранение переменной диапазона в области видимости, расположенной после конструкций `let`, `from` и `join`.

Выражение запроса, в котором за второй конструкцией `from` следует конструкция `select`

```
from x1 in e1
from x2 in e2
select v
```

преобразуется в

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

Выражение запроса, в котором за второй конструкцией `from` следует что-то иное, чем конструкция `select`

```
from x1 in e1
from x2 in e2
...
```

преобразуется в

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
```

Выражение запроса с конструкцией `let`

```
from x in e
let y = f
...
```

преобразуется в

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

Выражение запроса с конструкцией `where`

```
from x in e
where f
...
```

преобразуется в

```
from x in ( e ) . Where ( x => f )
...
```

Выражение запроса с конструкцией `join` без конструкции `into`, за которой следует конструкция `select`

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

преобразуется в

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

Выражение запроса с конструкцией `join` без конструкции `into`, за которой следует что-то иное, чем конструкция `select`

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

преобразуется в

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```

Выражение запроса, в котором присутствует конструкция `join` с конструкцией `into`, за которой следует конструкция `select`

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

преобразуется в

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

Выражение запроса, в котором присутствует конструкция `join` с конструкцией `into`, за которой следует что-то иное, чем конструкция `select`

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

преобразуется в

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 ,
    g })
```

Выражение запроса с конструкцией `orderby`

```
from x in e
orderby k1 , k2 , ... , kn
...
```

преобразуется в

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
... .
ThenBy ( x => kn )
...
```

Если задан порядок сортировки по убыванию, выполняется вызов `OrderByDescending` или `ThenByDescending`.

Следующие преобразования предполагают, что в выражении запроса конструкции `let`, `where`, `join` или `orderby` отсутствуют и задано не более одной начальной конструкции `from`.

Пример

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

преобразуется в

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c.Name, o.OrderID, o.Total })
```

Пример

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

преобразуется в

```
from * in customers.
SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

заключительное преобразование для которого будет

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

где x является идентификатором, сгенерированным компилятором, который иными способами невидим и недоступен.

Пример

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

преобразуется в

```
from * in orders.
   Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

заключительное преобразование для которого будет

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

где x является идентификатором, сгенерированным компилятором, который иными способами невидим и недоступен.

Пример

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

преобразуется в

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
(c, o) => new { c.Name, o.OrderDate, o.Total })
```

Пример

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

преобразуется в

```
from * in customers.
   GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
(c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

заключительное преобразование которого будет

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID, (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

где x и y являются идентификаторами, сгенерированными компилятором, которые иными способами невидимы и недоступны.

Пример

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

имеет заключительное преобразование

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

7.16.2.5. Конструкции select

Выражение запроса форме

```
from x in e select v
```

преобразуется в

```
( e ) . Select ( x => v )
```

за исключением случая, когда v является идентификатором x, то преобразование выглядит просто как

```
( e )
```

Например

```
from c in customers.Where(c => c.City == "London")
select c
```

преобразуется просто в

```
customers.Where(c => c.City == "London")
```

7.16.2.6. Конструкция GroupBy

Выражение запроса в форме

```
from x in e group v by k
```

преобразуется в

```
( e ) . GroupBy ( x => k , x => v )
```

за исключением случая, когда v является идентификатором x. Тогда

```
( e ) . GroupBy ( x => k )
```

Пример

```
from c in customers
group c.Name by c.Country
```

преобразуется в

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

7.16.2.7. Прозрачные идентификаторы

Некоторые преобразования вводят переменные диапазона с *прозрачными идентификаторами* (*transparent identifiers*), обозначаемыми *. Прозрачные иденти-

фикаторы в действительности не являются нормальной конструкцией языка; они существуют только на определенном шаге процесса преобразования выражения запроса.

Если в преобразование запроса вводится прозрачный идентификатор, в ходе дальнейших шагов преобразования его действие распространяется на анонимные функции и инициализаторы анонимных объектов. В этих контекстах прозрачные идентификаторы ведут себя следующим образом:

- Когда прозрачный идентификатор является параметром анонимной функции, элементы связанного с ним анонимного типа автоматически оказываются в области видимости в теле анонимной функции.
- Когда элемент с прозрачным идентификатором находится в области видимости, элементы этого элемента также находятся в области видимости.
- Когда прозрачный идентификатор является описателем элемента в инициализаторе анонимного объекта, он определяет элемент с прозрачным идентификатором.

На шагах преобразования, описанных выше, прозрачные идентификаторы всегда вводились вместе с анонимными типами с целью захватить несколько переменных диапазона как элементы единственного объекта. В реализации C# разрешается использовать вместо анонимных типов другой механизм группировки нескольких переменных диапазона. Следующие примеры преобразований предполагают использование анонимных типов и показывают, как могут быть преобразованы прозрачные идентификаторы.

Пример

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

преобразуется в

```
from * in customers.
   SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

и далее в

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

что, когда прозрачные идентификаторы удалены, эквивалентно

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

где `x` является идентификатором, сгенерированным компилятором, который иными способами невидим и недоступен.

Пример

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

преобразуется в

```
from * in customers.
   Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

и далее сокращается до

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

завершающее преобразование для которого будет

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID, (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID, (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

где `x`, `y`, и `z` — сгенерированные компилятором идентификаторы, которые иными способами невидимы и недоступны.

7.16.3. Паттерн выражения запроса

Паттерн выражения запроса (query expression pattern) устанавливает набор методов, который могут реализовывать типы для поддержки выражений запроса. Поскольку выражения запроса преобразуются в вызовы методов с помощью синтаксического отображения, есть значительная гибкость в том, как типы реализуют паттерн выражения запроса. Например, методы образца могут быть реализованы как методы экземпляра или как методы расширения, поскольку и те и другие имеют одинаковый синтаксис вызова, и эти методы могут вызывать делегатов или деревья выражений, поскольку анонимные функции преобразуются к обоим видам.

Рекомендуемая форма обобщенного типа `C<T>`, который поддерживает паттерн выражения запроса, представлена ниже. Обобщенный тип используется, чтобы показать действительное соотношение между типами параметров и результата, но можно реализовать паттерн и для необобщенных типов.

```
class C<T> : C
{
   public C<T> Where(Func<T,bool> predicate);

   public C<U> Select<U>(Func<T,U> selector);

   public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
      Func<T,U,V> resultSelector);
```

```
   public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
      Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

   public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
      Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

   public O<T> OrderBy<K>(Func<T,K> keySelector);

   public O<T> OrderByDescending<K>(Func<T,K> keySelector);

   public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

   public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
      Func<T,E> elementSelector);
}

class O<T> : C<T>
{
   public O<T> ThenBy<K>(Func<T,K> keySelector);

   public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
   public K Key { get; }
}
```

Приведенные выше методы используют обобщенные типы делегатов `Func<T1, R>` и `Func<T1, T2, R>`, но они точно так же могут использовать другие типы делегатов или типы деревьев выражений с тем же соотношением типов параметров и результата.

Отметим, что рекомендованное соотношение между `C<T>` и `O<T>`, которое обеспечивает применение методов `ThenBy` и `ThenByDescending` только к результату `OrderBy` или `OrderByDescending`. Также отметим, что рекомендованная форма результата `GroupBy` — последовательность последовательностей, в которой каждая внутренняя последовательность имеет добавочное свойство `Key`.

БИЛЛ ВАГНЕР

`ThenBy` чаще всего будет выполняться быстрее, чем `OrderBy`, поскольку в нем необходимо сортировать только внутренние последовательности, имеющие более одного значения.

Пространство имен `System.Linq` обеспечивает реализацию паттерна операции запроса для любого типа, реализующего интерфейс `System.Collections.Generic.IEnumerable<T>`.

БИЛЛ ВАГНЕР

Также существует реализация для любого типа, который реализует `IQueryable<T>`.

ЭРИК ЛИППЕРТ

Эта сигнатура для `Join` является одним из главных факторов части алгоритма выведения типов, которую можно назвать «нахождение границ и затем фиксация к наилучшей». Если внутренний ключ имеет, скажем, тип `int`, а внешний ключ — тип `int?`, то предпочтительнее было бы вместо того, чтобы выведение типов заканчивалось неудачно из-за «несоответствия», просто выбрать наиболее общий из двух типов. Поскольку любой `int` является также типом `int?`, алгоритм выведения типов должен выбрать `int?` для `K`.

7.17. Операции присваивания

Операции присваивания присваивают новое значение переменной, свойству, событию или элементу индексатора.

присваивание:
  унарное-выражение   операция-присваивания   выражение

операция-присваивания:
  =
  +=
  -=
  ***=**
  /=
  %=
  &=
  |=
  ^=
  <<=
  присваивание-со-сдвигом-вправо

Левый операнд операции присваивания должен быть выражением, классифицируемым как переменная, доступ к свойству, доступ к индексатору или доступ к событию.

Операция = называется **простой операцией присваивания**. Она присваивает значение правого операнда переменной, свойству или элементу индексатора, указанным в качестве левого операнда. Левый операнд простой операции присваивания описан в разделе 7.17.1.

Другие операции присваивания, кроме операции =, называются **сложными операциями присваивания**. Такие операции выполняют указанную операцию с двумя операндами, а затем присваивают полученное значение переменной, свойству или элементу индексатора, указанным в качестве левого операнда. Сложные операции присваивания описаны в разделе 7.17.2.

Операции += и -= с выражением доступа к событию в качестве левого операнда называются *операциями присваивания события*. Никакая другая операция присваивания не допускается по отношению к левому операнду в форме доступа к событию. Операции присваивания события описаны в разделе 7.17.3.

Операции присваивания являются правоассоциативными, это означает, что действия группируются справа налево. Например, выражение в форме `a = b = c` вычисляется как `a = (b = c)`.

7.17.1. Простое присваивание

Операция `=` называется операцией простого присваивания.

Если левый операнд простого присваивания представлен в форме `E.P` или `E[Ei]`, где `E` имеет тип времени компиляции `dynamic`, то операция присваивания является динамически связанной, и разрешение, описанное ниже, будет происходить во время выполнения программы в соответствии с фактическим типом `E`.

Для простого присваивания правый операнд должен быть выражением, которое неявно преобразуется к типу левого операнда. Эта операция присваивает значение правого операнда переменной, свойству или элементу индексатора, указанным в качестве левого операнда.

Результатом выражения простого присваивания является значение, присвоенное левому операнду. Результат имеет тот же тип, что и левый операнд, и всегда является значением.

Если левый операнд является свойством или доступом к индексатору, это свойство или этот индексатор должны иметь код доступа `set`. Если это не так, выдается ошибка времени связывания.

Обработка операции простого присваивания во время выполнения программы состоит из следующих шагов:

- Если `x` классифицируется как переменная:
  - Вычисляется `x`, чтобы получить переменную.
  - Вычисляется `y` и при необходимости приводится к типу `x` с помощью неявного преобразования (раздел 6.1).
  - Если переменная, заданная `x`, является элементом массива *ссылочного типа*, во время выполнения программы выполняется проверка, позволяющая убедиться, что значение, вычисленное для `y`, совместимо с экземпляром массива, элементом которого является `x`. Проверка проходит успешно, если `y` есть `null` или существует неявное ссылочное преобразование (раздел 6.1.6) для фактического типа экземпляра, на который ссылается `y`, к фактическому типу элемента экземпляра массива, содержащего `x`. В противном случае выбрасывается исключение `ArrayTypeMismatchException`.
  - Значение, полученное в результате вычисления и приведения `y`, сохраняется в том месте, которое задает вычисление `x`.
- Если `x` классифицируется как свойство или доступ к индексатору:
  - Вычисляются связанные с `x` выражение экземпляра (если `x` не является `static`) и список аргументов (если `x` является доступом к индексатору), и результаты используются при последующих вызовах кода доступа `set`.
  - Вычисляется `y` и при необходимости приводится к типу `x` с помощью неявного преобразования (раздел 6.1).

- Вызывается код доступа `set` для `x` со значением аргумента `value`, являющимся значением, вычисленным для `y`.

Правила ковариантности массивов (раздел 12.5) допускают, чтобы значение типа массива `A[]` являлось ссылкой на экземпляр типа массива `B[]` при условии, что существует неявное ссылочное преобразование из `B` к `A`. В соответствии с этими правилами при операции присваивания значения элементу массива ссылочного типа требуется проверка во время выполнения программы, позволяющая убедиться, что значение, которое должно быть присвоено, совместимо с экземпляром массива. В примере:

```
string[] sa = new string[10];
object[] oa = sa;
oa[0] = null;              // Допустимо
oa[1] = "Hello";           // Допустимо
oa[2] = new ArrayList();   // ArrayTypeMismatchException
```

последнее присваивание приводит к исключению `System.ArrayTypeMismatchException`, поскольку экземпляр массива `ArrayList` не может храниться в элементе типа `string[]`.

> **БИЛЛ ВАГНЕР**
>
> Это утверждение означает, что присваивание массива не копирует массив, а добавляет новую ссылку на то же самое место хранения.

Когда целевым объектом присваивания является свойство или индексатор, объявленный в *структуре*, выражение экземпляра, связанное с доступом к свойству или индексатору, должно быть классифицировано как переменная. Если выражение экземпляра классифицировано как значение, выдается ошибка времени связывания. В соответствии с разделом 7.6.4 то же самое правило применимо к полям.

Для объявлений:

```
struct Point
{
   int x, y;
   public Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   public int X
   {
      get { return x; }
      set { x = value; }
   }

   public int Y
   {
      get { return y; }
      set { y = value; }
   }
}
```

```
struct Rectangle
{
   Point a, b;

   public Rectangle(Point a, Point b)
   {
      this.a = a;
      this.b = b;
   }

   public Point A
   {
      get { return a; }
      set { a = value; }
   }

   public Point B
   {
      get { return b; }
      set { b = value; }
   }
}
```

в примере

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

операции присваивания для `p.X`, `p.Y`, `r.A` и `r.B` допустимы, поскольку `p` и `r` являются переменными. Однако в примере

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

все присваивания недопустимы, поскольку `r.A` и `r.B` не являются переменными.

ДЖОЗЕФ АЛЬБАХАРИ

В ранней версии компилятора C# 1.0 допускались такие операции присваивания, как `r.A.X = 10`, но они «молча» завершались неудачно, поскольку `r.A` возвращает *копию* `Point` (то есть значение), а не переменную. Пользователи находили такое поведение странным, поэтому это условие было сочтено ошибкой.

БИЛЛ ВАГНЕР

В данном обсуждении названа еще одна причина для того, чтобы структуры были неизменяемыми.

7.17.2. Сложное присваиваивание

Если левый операнд сложного присваивания представлен в форме `E.P` или `E[Ei]`, где `E` имеет тип времени компиляции `dynamic`, то операция присваивания является динамически связанной (раздел 7.2.2). В этом случае во время компиляции для выражения присваивания будет определен тип `dynamic`, и разрешение, описанное ниже, будет происходить во время выполнения программы в соответствии с фактическим типом `E`.

Операция в форме `x` *op*`= y` обрабатывается с применением разрешения перегрузки бинарной операции (раздел 7.3.4), как операция, записанная в виде `x` *op* `y`. Тогда:

- Если тип возвращаемого значения выбранной операции *неявно* приводится к типу `x`, операция вычисляется как `x = x` *op* `y`, но `x` вычисляется только единожды.
- Иначе, если выбранная операция является предопределенной и если тип возвращаемого значения *явно* преобразуется к типу `x` или если это операция сдвига, то операция вычисляется как `x = (T)(x` *op* `y)`, где `T` является типом `x`, но `x` вычисляется только единожды.
- Иначе операция сложного присваивания недопустима и выдается ошибка времени связывания.

ПИТЕР СЕСТОФТ

Указанное во втором пункте условие, что `y` должно быть неявно преобразуемо к типу `x`, означает, что на сложные операции в C# наложены большие ограничения, нежели в C, C++ и Java. В этих языках присваивание `x+=0.9` считается законным, даже если переменная `x` имеет целый тип. Если `x` неотрицательно, это присваивание не произведет никакого действия; если же `x` отрицательно, к `x` добавится `1`. В C# такое присваивание запрещено — и на мой взгляд, это разумно.

ВЛАДИМИР РЕШЕТНИКОВ

Забавное следствие этих правил состоит в том, что выражение `x <<= null` допустимо, когда переменная `x` имеет тип `int`. Конечно, это всегда приводит к генерации исключения во время выполнения программы.

Выражение «вычисляется только единожды» означает, что при вычислении `x` *op* `y` результаты некоторой составной части выражения `x` на какое-то время сохраняются и затем вновь используются, когда происходит присваивание значения `x`. Например, в операции присваивания `A()[B()] += C()`, где `A` является методом, возвращающим `int[]`, а `B` и `C` являются методами, возвращающими `int`, эти методы вызываются только один раз в порядке: `A`, `B`, `C`.

Когда левый операнд сложного присваивания является доступом к свойству или к индексатору, это свойство или этот индексатор должны иметь оба кода доступа `get` и `set`. Если это не так, выдается ошибка времени связывания.

Второе из вышеприведенных правил позволяет в определенных контекстах вычислять x op= y как x = (T)(x op y). Это правило выполняется таким образом, что предопределенные операции могут использоваться как сложные операции, когда левый операнд имеет тип sbyte, byte, short, ushort или char. Даже когда оба аргумента имеют один из вышеперечисленных типов, предопределенная операция дает результат типа int, как описано в разделе 7.3.6.2. Таким образом, без приведения невозможно присвоить значение результата левому операнду.

Интуитивно понятное действие этого правила заключается просто в том, что x *op*= y допустимо тогда, когда оба выражения x *op* y и x = y допустимы. В примере

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;         // Допустимо
b += 1000;      // Ошибка: b = 1000 недопустимо
b += i;         // ошибка: b = i недопустимо
b += (byte)i;   // Допустимо

ch += 1;        // Ошибка: ch = 1 недопустимо
ch += (char)1;  // Допустимо
```

интуитивно понятная причина каждой ошибки заключается в том, что соответствующее простое присваивание также является ошибочным.

Это также означает, что сложные операции присваивания поддерживают расширенные операции. В примере

```
int? i = 0;
i += 1;         // Допустимо
```

используется расширенная операция +(int?,int?).

7.17.3. Присваивание событий

Если левый операнд операции классифицируется как доступ к событию, то выражение вычисляется следующим образом:

- Вычисляется выражение экземпляра доступа к событию, если таковое существует.
- Вычисляется правый операнд операции += или -= и при необходимости приводится к типу левого операнда при помощи неявного преобразования (раздел 6.1).
- Вызывается код доступа к событию со списком аргументов, содержащим правый операнд после его вычисления, и, возможно, приведения. Если речь идет об операции +=, вызывается код доступа add; если же имеется в виду операция -=, то вызывается код доступа remove.

Выражение присваивания для события не возвращает значения. Таким образом, оно допустимо только в контексте *оператора-выражения* (раздел 8.6).

7.18. Выражения

Выражение может быть либо *выражением-без-присваивания*, либо *присваиванием*.

выражение
  выражение-без-присваивания
  присваивание

выражение-без-присваивания:
  условное-выражение
  лямбда-выражение
  выражение-запроса

7.19. Константные выражения

Константными-выражениями являются выражения, которые могут быть полностью вычислены во время компиляции.

константное-выражение:
  выражение

Константное выражение должно быть литералом `null` или значением одного из следующих типов: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string` или любого перечислимого типа. В константных выражениях допустимы только следующие конструкции:

- Литералы (в том числе литерал `null`).
- Ссылки на `const` элементы классов или структур.
- Ссылки на элементы перечислений.
- Ссылки на `const` параметры или локальные переменные.
- Подвыражения в круглых скобках, которые сами являются константными выражениями.
- Выражения приведения, при условии, что тип целевого объекта является одним из типов, представленных выше.
- Выражения `checked` и `unchecked`.
- Выражения значений по умолчанию.
- Предопределенные унарные операции `+`, `–`, `!` и `~`.
- Предопределенные бинарные операции `+`, `–`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=` и `>=`, при условии, что каждый их операнд имеет тип из списка, представленного выше.
- Условная операция `?:`.

В константных выражениях допустимы следующие преобразования:

- Тождественные преобразования.
- Арифметические преобразования.

- Преобразования перечислений.
- Преобразования константных выражений.
- Неявные и явные ссылочные преобразования, при условии что исходным для этих преобразований является константное выражение, которое при вычислении дает значение `null`.

Иные преобразования, включая упаковку, распаковку и неявные ссылочные преобразования с ненулевыми значениями, в константных выражениях недопустимы. Например:

```
class C {
   const object i = 5;          // Ошибка: преобразование упаковки не разрешено
   const object str = "hello";  // Ошибка: неявное ссылочное преобразование
}
```

В данном примере инициализация `i` является ошибкой, поскольку требуется преобразование упаковки. Инициализация `str` является ошибкой, поскольку требуется неявное ссылочное преобразование из ненулевого значения.

Во всех случаях, когда выражение соответствует требованиям, указанным выше, оно вычисляется во время компиляции. Это верно, даже если выражение является подвыражением большего выражения, содержащего неконстантные конструкции.

Для вычисления константных выражений во время компиляции используются те же самые правила, что и для вычисления неконстантных выражений во время выполнения программы. Разница в том, что вычисление во время выполнения программы может порождать исключения, тогда как вычисление выражений во время компиляции может приводить к ошибке компиляции.

Если константное выражение не помещено явно в непроверяемый контекст, переполнения, которые происходят при целочисленных арифметических операциях и преобразованиях при вычислении выражений во время компиляции, всегда приводят к ошибкам компиляции (раздел 7.19).

Константные выражения существуют в контекстах, представленных ниже. В этих контекстах будет появляться ошибка компиляции, если выражение не может быть полностью вычислено во время компиляции.

- Объявления констант (раздел 10.4).
- Объявления элементов перечисления (раздел 14.3).
- Метки `case` оператора `switch` (раздел 8.7.2).
- Операторы `goto case` (раздел 8.9.3).
- Величины размерностей в выражении создания массива (7.6.10.4), которые содержат инициализатор.
- Атрибуты (глава 17).

При неявных преобразованиях константных выражений (раздел 6.1.8) константные выражения типа `int` могут быть приведены к типам `sbyte`, `byte`, `short`, `ushort`, `uint` и `ulong` при условии, что значение константого выражения находится в пределах интервала значений нужного типа.

7.20. Булевские выражения

Булевским-выражением является выражение, которое возвращает результат типа `bool` либо непосредственно, либо в результате выполнения операции `operator true` в некотором контексте, как описано далее.

булевское-выражение:
  выражение

Булевское выражение является управляющим условным выражением *оператора-if* (раздел 8.7.1), *оператора-while* (раздел 8.8.1), *оператора-do* (раздел 8.8.2) и *оператора-for* (раздел 8.8.3). Управляющее условное выражение для операции `?:` (раздел 7.14) подчиняется тем же самым правилам, что и булевское выражение, но в соответствии с приоритетами операций оно классифицируется как *условное-выражение-ИЛИ*.

Булевское выражение должно быть неявно преобразуемо к типу `bool` или к типу, который реализует `operator true`. Если ни одно из этих требований не выполняется, выдается ошибка времени связывания.

Если булевское выражение нельзя неявно преобразовать к типу `bool`, но оно реализует `operator true`, то для вычисления выражения используется реализация операции `operator true` соответствующего типа, и в результате получается значение типа `bool`.

Примером типа, реализующего `operator true` и `operator false`, может служить структурный тип `DBBool`, описанный в разделе 11.4.2.

Глава 8
Операторы

В языке C# существуют различные виды операторов. Большинство из них знакомо разработчикам, которые ранее программировали на C или C++.

```
оператор:
   оператор-объявления

вложенный-оператор:
   пустой-оператор
   оператор-выбора
   оператор-перехода
   оператор-checked
   оператор-lock
   оператор-yield
```

Вложенный оператор можно использовать внутри других операторов, причем степень вложенности не ограничена. Использование *вложенного-оператора*, а не *оператора* исключает использование в этом контексте операторов объявлений и помеченных операторов. В примере:

```
void F(bool b)
{
   if (b)
      int i = 44;
}
```

мы получим ошибку компиляции, потому что для ветви `if` требуется *вложенный-оператор*, а не *оператор*. Если бы данный код был допустимым, то переменная `i` объявлялась, но ее нельзя было бы использовать. Однако если переменную `i` объявить в блоке, то ошибки компиляции не будет.

8.1. Конечные точки и достижимость

Каждый оператор имеет **конечную точку**. Интуитивно понятно, что это место в коде, которое следует непосредственно за данным оператором. Правила выполнения сложных операторов, которые содержат в себе другие операторы, определяют действие, которое будет выполнено при достижении конечной точки вложенного оператора. Например, при достижении конечной точки оператора в блоке управление передается на следующий оператор в этом блоке.

Если на оператор в принципе может быть передано управление при выполнении программы, то говорят, что этот оператор **достижимый**. Если оператор никогда не может быть выполнен, его называют **недостижимым**.

В примере

```
void F()
{
   Console.WriteLine("reachable");
   goto Label;
   Console.WriteLine("unreachable");
   Label:
   Console.WriteLine("reachable");
}
```

второй вызов `Console.WriteLine` является недостижимым, потому что выполнить этот оператор нет никакой возможности.

При обнаружении в коде недостижимого оператора компилятор выдает предупреждение. Недостижимый оператор не является ошибкой.

Чтобы определить, являются ли оператор или его конечная точка достижимыми, компилятор выполняет анализ потока управления согласно правилам достижимости, которые определены для каждого типа операторов. Анализ потока управления учитывает значения константных выражений (раздел 7.19), которые управляют выполнением операторов, но возможные значения неконстантных выражений не вычисляются. Иными словами, для анализа потока управления неконстантное выражение некоторого типа может иметь любое допустимое для данного типа значение.

В примере

```
void F()
{
   const int i = 1;
   if (i == 2) Console.WriteLine("unreachable");
}
```

логическое выражение в операторе `if` является константным, так как оба операнда, участвующие в операции проверки равенства ==, являются константами. Так как константные значения вычисляются во время компиляции, данное условие всегда имеет значение `false`, и оператор `Console.WriteLine` считается недостижимым.

Однако, если в приведенном примере мы поменяем `i` на локальную переменную:

```
void F()
{
   int i = 1;
   if (i == 2) Console.WriteLine("reachable");
}
```

Оператор `Console.WriteLine` будет считаться достижимым, хотя на самом деле никогда не будет выполняться.

> **ЭРИК ЛИППЕРТ**
>
> Есть много других выражений, о которых человеку ясно, что они всегда равны `false`, однако они не являются равными `false` для анализа потока управления. Например, если заменить условие в операторе `if` (см. пример выше) выражением `(i*0 == 0)`, то оператор `Console.WriteLine` будет считаться достижимым, хотя абсолютно ясно, что на самом деле он недостижим.

Блок функционального элемента всегда считается достижимым. Путем успешного применения правил достижимости для каждого оператора в блоке можно определить достижимость любого оператора.

Например:

```
void F(int x)
{
   Console.WriteLine("start");
   if (x < 0) Console.WriteLine("negative");
}
```

достижимость второго `Console.WriteLine` определяется следующим образом:

- Первый оператор `Console.WriteLine` считается достижимым, потому что тело метода `F` достижимо.
- Конечная точка первого оператора `Console.WriteLine` достижима, потому что сам оператор является достижимым.
- Оператор `if` достижим, потому что конечная точка первого `Console.WriteLine` достижима.
- Второй оператор `Console.WriteLine` достижим, потому что выражение в операторе `if` не имеет константного значения `false`.

Есть две ситуации, когда выдается ошибка компиляции, связанная с достижимостью конечной точки оператора:

- Поскольку в операторе `switch` не допускается, чтобы после выполнения одной `switch` секции естественным образом выполнялась следующая, достижимость конечной точки списка операторов секции `switch` является ошибкой компиляции. Такая ошибка обычно говорит о том, что пропущен оператор `break`.
- Если конечная точка блока функционального элемента, возвращающего значение, достижима, это является ошибкой компиляции. Обычно такая ошибка возникает, если пропущен оператор `return`.

БИЛЛ ВАГНЕР

Правила определения достижимости могут определить фактически недостижимый код как достижимый, но не наоборот. В этом примере мы не получим никаких предупреждений о недостижимости кода:

```
public class Program
{
   public static int counter = 5;

   static void Main(string[] args)
   {
      if (counter == 6)
         Console.WriteLine("weird");
      else
         Console.WriteLine("normal");
   }
}
```

продолжение ⇨

> Очевидно, что «`weird`» никогда не будет выведено на консоль, но правила языка работают по другим принципам.

8.2. Блоки

Блок позволяет использовать несколько операторов там, где предусмотрен только один.

блок:
   `{` *список операторов*$_{opt}$ `}`

Блок состоит из необязательного *списка-операторов* (раздел §8.2.1), заключенных в фигурные скобки. Если список операторов отсутствует, блок называют пустым.

Блок может содержать *операторы-объявления* (раздел 8.5). Областью действия локальной переменной или константы, объявленной в блоке, является этот блок.

Внутри блока значение имени, используемого в контексте выражения, всегда должно быть одним и тем же.

Блок выполняется следующим образом:

- Если блок пустой, управление передается на конечную точку блока.
- Если блок не пустой, управление передается на список операторов. Когда и если управление передается на конечную точку списка операторов, управление передается на конечную точку блока.

Список операторов блока достижим, если достижим сам блок.

Конечная точка блока достижима, если блок пустой или если достижима конечная точка списка операторов.

Блок, который содержит один или более операторов `yield` (раздел 8.14), называется блоком итератора. Блоки итератора используются для реализации функциональных элементов как итераторов (раздел 10.14). К блокам итераторов применимы некоторые дополнительные ограничения:

- Появление оператора `return` в блоке итератора приводит к ошибке компиляции (но операторы `yield return` допускаются).
- Появление небезопасного контекста в блоке итератора приводит к ошибке компиляции (раздел 18.1). Блок итератора всегда определяет безопасный контекст, даже если его объявление находится внутри небезопасного контекста.

8.2.1. Списки операторов

Список операторов состоит из одного или более операторов, записанных последовательно. Списки операторов используются в *блоках* (раздел 8.2) и *switch-блоках* (раздел 8.7.2).

список-операторов:
  оператор
  список-операторов оператор

Список операторов выполняется путем передачи управления на его первый оператор. Когда и если управление передается на конечную точку оператора, управление передается на следующий оператор. Когда и если управление передается на конечную точку последнего оператора, управление передается на конечную точку списка операторов.

Оператор в списке операторов является достижимым, если выполняется хотя бы одно из следующих условий:

- Оператор является первым в списке операторов и сам список достижим.
- Конечная точка предыдущего оператора достижима.
- Оператор является помеченным оператором, и на метку ссылается достижимый оператор `goto`.

ВЛАДИМИР РЕШЕТНИКОВ

Это правило не применяется, когда оператор `goto` находится внутри блоков `try` или `catch` оператора `try`, который содержит блок `finally`, а помеченный оператор находится вне оператора `try`, и конечная точка блока `finally` недостижима. Например:

```
class C
{
   static void Main()
   {
      int x;
      try
      {
         goto A; // Достижимый оператор
      }
      finally
      {
         throw new System.Exception();
      }
      A: x.ToString(); // Недостижимый оператор
   }
}
```

Конечная точка списка операторов достижима, если достижима конечная точка последнего оператора из списка.

8.3. Пустой оператор

Пустой оператор не выполняет никаких действий.

пустой-оператор:
  ;

Пустой оператор используется там, где не требуется выполнять действий, но наличие оператора требуется по контексту.

ДЖОН СКИТ

Учитывая, как легко бывает пропустить пустой оператор при чтении кода программы, думаю, что в качестве пустого оператора можно было бы использовать более удобную конструкцию. Данный код является допустимым, но, вероятно, работает не так, как задумано:

```
while (text.IndexOf("xx") != -1);
{
   text = text.Replace("xx", "x");
}
```

Пустой оператор в цикле `while` достаточно трудно увидеть. Если бы в качестве пустого оператора можно было использовать что-то типа `void;`, то компилятор мог бы определять такие описки как ошибки.

ДЖЕСИ ЛИБЕРТИ

Пример, который привел Джон, показывает нам, почему использовать пустые операторы нежелательно. На них необходимо акцентировать внимание, и не только с помощью комментариев. Проблему в его коде можно обострить, разместив пустой оператор на новой строке или, что еще лучше, внутри блока операторов:

```
while (text.IndexOf("xx") != -1)
{
   ;
}
{
   text = text.Replace("xx", "x");
}
```

Теперь все обратят внимание, что цикл `while` содержит пустой оператор. Использование следующего блока операторов также необязательно, но отступы позволяют избежать путаницы.

Выполнение пустого оператора просто передает управление на его конечную точку. Таким образом, конечная точка пустого оператора считается достижимой, если достижим сам оператор. Имеет смысл использовать пустой оператор в цикле `while`, у которого отсутствует тело цикла:

```
bool ProcessMessage() {...}
void ProcessMessages()
{
   while (ProcessMessage())
      ;
}
```

Также пустой оператор можно использовать для метки, которая находится перед закрывающей скобкой в блоке операторов:

```
void F()
{
...
   if (done) goto exit;
...
   exit: ;
}
```

8.4. Помеченные операторы

Помеченный-оператор допускает, чтобы оператору предшествовала метка. Помеченные операторы можно использовать в блоке, но они не должны использоваться как вложенные операторы.

```
помеченный-оператор:
   идентификатор  :  оператор
```

Помеченный оператор объявляет метку с именем, определяемым *идентификатором*. Область видимости метки – весь блок, в котором она объявлена, включая все вложенные блоки. Если области видимости двух меток пересекаются, компилятор выдаст ошибку.

На метку может ссылаться оператор `goto` (раздел 8.9.3), если он находится в области видимости метки. Из этого факта следует, что оператор `goto` может передавать управление в пределах блока или за его пределы, но никогда не внутрь блока.

Метки объявляются в отдельном пространстве имен и не пересекаются с другими идентификаторами. Данный пример является допустимым:

```
int F(int x)
{
   if (x >= 0) goto x;
   x = -x;
   x: return x;
}
```

Идентификатор `x` можно использовать и для имени метки, и для имени параметра.

Выполнение метки сводится к выполнению оператора, следующего за меткой.

В дополнение к алгоритму расчета достижимости по нормальному потоку управления метка считается достижимой, если достижим оператор `goto`, ссылающийся на нее. (Исключением является случай, когда `goto` находится внутри оператора `try`, имеющего блок `finally`, метка находится за пределами `try`, а конечная точка блока `finally` является недостижимой. В этом случае метка вне оператора `try` является недостижимой для `goto`.)

ЭРИК ЛИППЕРТ

Например, блок `finally` может всегда выбрасывать исключение. Тогда в программе будет достижимый оператор `goto` и потенциально недостижимая метка, на которую он ссылается.

ПИТЕР СЕСТОФТ

Более полно эта тема освещена в примечаниях к разделу 8.9.3 при обсуждении оператора `goto`.

8.5. Операторы объявления

Оператор-объявления объявляет локальную переменную или константу.

```
оператор-объявления:
   объявление-локальной-переменной   ;
   объявление-локальной-константы    ;
```

8.5.1. Объявление локальных переменных

Объявление локальной переменной может содержать объявление одной или нескольких переменных:

```
объявление-локальной-переменной:
   тип-локальной-переменной    описатели-локальной-переменной

тип-локальной-переменной:
   тип
   var

описатели-локальной-переменной:
   описатель-локальной-переменной
   описатели-локальной-переменной   ,   описатель-локальной-переменной

описатель-локальной-переменной:
   идентификатор

идентификатор   =   инициализатор-локальной-переменной

инициализатор-локальной-переменной:
   выражение
   инициализатор-массива
```

Тип-локальной-переменной в *объявлении-локальной-переменной* либо определяет тип непосредственно, либо с помощью ключевого слова `var` указывает, что тип необходимо вывести на основе инициализатора. За типом следует список *описателей-локальной-переменной,* каждый из которых вводит новую переменную. *Описатель-локальной-переменной* состоит из *идентификатора*, являющегося именем переменной, и необязательного *инициализатора-локальной-переменной*, который следует за знаком «=» и задает начальное значение переменной.

При объявлении переменных `var` является контекстно-зависимым ключевым словом (раздел 2.4.3). Если *тип-локальной-переменной* объявлен как `var` и в пределах области действия нет типа с именем `var`, то это — **объявление неявно типизированной локальной переменной**. Ее тип будет определен по типу выражения инициализатора для этой переменной. Объявление неявно типизированной локальной переменной имеет следующие ограничения:

- Объявление не может содержать несколько *описателей-локальной-переменной.*
- Объявление должно содержать *инициализатор локальной-переменной.*
- Инициализатор должен быть *выражением.*

- *Выражение* инициатора должно иметь тип времени компиляции.
- *Выражение* инициатора не может ссылаться на саму объявляемую переменную.

> **ЭРИК ЛИППЕРТ**
>
> В первых планах реализации этой возможности хотели сделать возможным несколько описателей.
>
> `var a = 1, b = 2.5;`
>
> Половина разработчиков C#, увидевших этот код, посчитали, что семантика должна соответствовать
>
> `double a = 1, b = 2.5;`
>
> Другая половина считала, что семантика следующая:
>
> `int a = 1; double b = 2.5;`
>
> При этом каждая сторона считала свою интерпретацию «интуитивно понятной». Когда реализация предполагает два абсолютно противоположных «интуитивно понятных» решения, то лучше вообще не реализовывать такую возможность (чтобы избежать проблем недопонимания).

> **КРИС СЕЛЛС**
>
> Я считаю, что несколько описателей переменных в одном объявлении только для того, чтобы не писать повторно имя типа, тоже должно быть недопустимым.

> **ЭРИК ЛИППЕРТ**
>
> Это ограничение сильно отличается от объявления явно типизированных переменных, для которых можно в инициализаторе ссылаться на него самого.
>
> Например, `int j=M(out j);` выглядит странно, но этот код является допустимым. Если бы это выражение выглядело как `var j=M(out j)`, то разрешение перегрузки не смогло бы определить тип, возвращаемый методом `M`, а следовательно, и тип переменной `j`, пока не станет известным тип аргумента. Естественно, ведь именно тип входного аргумента мы и пытаемся определить. Чтобы не решать, «что было раньше — курица или яйцо», спецификация просто не разрешает такую конструкцию.

Приведем несколько примеров неверного объявления неявно типизированных локальных переменных:

```
var x;              // Ошибка: нет инициализатора, чтобы по нему определить тип
var y = {1, 2, 3};  // Ошибка: инициализатор массива не допустим
var z = null;       // Ошибка: null не имеет типа
var u = x => x + 1; // Ошибка: анонимная функция не имеет типа
var v = v++;        // Ошибка: инициализатор не может ссылаться на саму переменную
```

Значение локальной переменной устанавливается с помощью выражения, использующего *простое-имя* (раздел 7.6.2), значение локальной переменной можно изменять с помощью *присваивания* (раздел 7.17). Локальная переменная должна быть явно присвоена (раздел 5.3) в каждом месте, где ее значение получают.

КРИС СЕЛЛС

Мне страшно нравятся неявно типизированные локальные переменные, когда их тип анонимный (в этом случае вам приходится их использовать) или когда тип переменной очевиден по типу ее инициализатора (а не потому, что вам лень в коде напечатать тип)! Например:

```
var a = new { Name = "Bob", Age = 42 }; // Хорошо
var b = 1;                              // Хорошо
var v = new Person();                   // Хорошо
var d = GetPerson();                    // ПЛОХО!
```

Компилятор удовлетворяется синтаксисом в объявленной переменной `d`, но мне жаль тех, кому придется читать такой код.

ЭРИК ЛИППЕРТ

В целом я согласен с предыдущим высказыванием, но хочу сделать одно замечание: `var` уместно использовать, если программист хочет донести мысль о том, что тип переменной не так важен. Важен ее смысл, а не детали реализации. Например, я часто пишу код так: `var attributes=ParseMethodAttributes();`. Этим я хочу сказать, что не так важно, возвращается `AttributeSyntax[]` или `List<AttributeSyntax>`. Важно, что коллекция атрибутов разобрана.

БИЛЛ ВАГНЕР

Признаюсь, что я часто использую объявление неявно типизированной переменной с помощью `var`. На самом деле я обычно использую `var` почти везде, кроме простых типов. Считаю, что семантическое представление переменной более важно, чем синтаксическое. В примере Криса метод `GetPerson()` может возвращать класс `Person` или класс, реализующий интерфейс `IPerson`, или что-то унаследованное от класса `Person` или реализации `IPerson`. В любом случае для меня не столь важна подобная неопределенность. Я понимаю идею, что в этой переменной есть какая-то информация (о человеке), и мне не важен точный тип переменной.

Область видимости локальной переменной, объявленной в *объявлении-локальной-переменной* — тот блок, внутри которого она объявлена. Попытка обратиться к локальной переменной в тексте, расположенном до *описателя-локальной-переменной*, вызовет ошибку компиляции. При попытке объявить другую локальную переменную или константу с тем же именем в области видимости локальной переменной также выдается ошибка компиляции.

Объявление нескольких локальных переменных в одном операторе эквивалентно нескольким объявлениям локальных переменных одного и того же типа. Инициализатор в объявлении локальной переменной эквивалентен использованию оператора присваивания сразу после объявления переменной. Например:

```
void F()
{
   int x = 1, y, z = x * 2;
}
```

точно соответствует следующему коду:

```
void F()
{
   int x; x = 1;
   int y;
   int z; z = x * 2;
}
```

При объявлении переменной с помощью `var` ее тип будет определен по типу выражения, которым она инициируется. Например:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

Объявление этих неявно типизированных переменных в точности эквивалентно объявлению следующих явно типизированных переменных:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

ПИТЕР СЕСТОФТ

В языке C# можно объявить локальные константы, но нельзя объявлять доступные только для чтения локальные переменные или параметры метода. Это отличается от конструкции `var` в языках `StandartML` и `Scala`, конструкции `let` в F# и `final` в Java. Будучи старым программистом функциональной парадигмы, я нахожу это неудобным. Зачастую мне нужно просто дать имя какому-то значению, а не объявлять изменяемую переменную. Оператор `using` (раздел 8.13) позволяет мне объявить неизменяемую локальную переменную, но это громоздко и неестественно и поэтому только путает большинство программистов. Кроме того, этот вариант работает только для локальных переменных и не может быть использован для параметров метода.

8.5.2. Объявление локальных констант

Объявление-локальных-констант описывает одну и более локальных констант.

```
объявление-локальной-константы:
   const   тип   описатели-констант

описатели-констант:
   описатель-константы
   описатели-констант   ,   описатель-константы

описатель-константы:
   идентификатор   =   константное-выражение
```

Тип в *объявлении-локальных-констант* задает тип констант, вводимых объявлением. За типом указывается список *описателей-констант*, каждый из

которых вводит новую константу. *Описатель-константы* состоит из *идентификатора*, который является именем константы, за которым следует символ «=» и *константное-выражение* (раздел 7.19), которое определяет значение константы.

Тип и *константное-выражение* в объявлении константы должны удовлетворять тем же правилам, что и при объявлении константного элемента класса (раздел 10.4).

Значение локальной константы получают в выражениях с использованием *простого-имени* (раздел 7.6.2).

Областью видимости локальной константы является блок, в котором она объявлена. Попытка обратиться к локальной константе из текста программы, расположенного до *описателя-константы*, вызовет ошибку компиляции. При объявлении другой локальной константы или переменной с тем же именем в области видимости константы также возникает ошибка компиляции.

Объявление нескольких локальных констант в одном операторе эквивалентно нескольким объявлениям констант с одним и тем же типом.

8.6. Операторы-выражения

Оператор-выражение вычисляет это выражение. Значение, вычисленное выражением (если оно есть), игнорируется.

```
оператор-выражение:
   выражение-оператора   ;

выражение-оператора:
   вызов-метода
   создание-объекта
   присваивание
   пост-инкрементное-выражение
   пост-декрементное-выражение
   пре-инкрементное-выражение
   пре-декрементное-выражение
```

Не все выражения допустимо использовать как операторы. Например, выражения `x+y` и `x==1`, которые просто вычисляют некоторое значение (которое будет игнорировано), в качестве операторов не допускаются.

Выполнение *оператора-выражения* вычисляет выражение и передает управление в его конечную точку. Конечная точка *оператора-выражения* достижима, если достижим сам *оператор-выражение*.

ДЖОН СКИТ

Иногда кто-нибудь предлагает разрешить объявлять бесконечные методы, то есть такие, которые никогда не заканчиваются нормально. При таком подходе единственный способ прервать выполнение метода – выбросить исключение. Очевидный пример подобного метода — это `Assert.Fail` в библиотеке unit-тестов.

Если это реализовать, конечная точка операторов-выражений, в которых выполняется вызов такого метода, будет недостижима. При необходимости можно было бы избежать фиктивного возврата значения или выбрасывания исключения, которые никогда не будут выполнены. Но я думаю, что в терминах и языка, и реализации овчинка выделки не стоит.

ЭРИК ЛИППЕРТ

Комиссия, отвечающая за стандарт языка ECMAScript, когда-то предлагала добавить тип «никогда», похожий на то, о чем говорит Джон. Я согласен, что иметь такую возможность в C# было бы хорошо, но затраты сил на ее реализацию слишком велики по сравнению с полученными удобствами.

8.7. Операторы выбора

Операторы выбора позволяют выполнить один из возможных операторов, исходя из значения некоторого выражения.

операторы-выбора:
  оператор-if
  оператор-switch

8.7.1. Оператор if

Оператор `if` выбирает оператор для выполнения по значению логического выражения.

оператор-if:
  `if` **(** *логическое-выражение* **)** *вложенный-оператор*
  `if` **(** *логическое-выражение* **)** *вложенный-оператор* **else** *вложенный-оператор*

При этом `else`-часть оператора относится к ближайшему (по правилам языка) оператору `if`. Таким образом, оператор `if`

```
if (x) if (y) F(); else G();
```

эквивалентен

```
if (x)
{
   if (y)
   {
      F();
   }
   else
   {
      G();
   }
}
```

Оператор `if` выполняется следующим образом:

- Вычисляется *логическое-выражение* (раздел 7.20).
- Если оно имеет значение `true`, управление передается первому вложенному оператору. По достижении его конечной точки управление передается на конечную точку оператора `if`.
- Если же логическое выражение имеет значение `false` и присутствует часть `else` оператора, то управление передается второму вложенному оператору. При достижении его конечной точки управление передается на конечную точку оператора `if`.
- Если результат логического выражения `false` и `else`-части нет, то управление передается на конечную точку оператора `if`.

Первый вложенный оператор оператора `if` является достижимым, если достижим сам оператор `if` и логическое выражение не является константным значением `false`.

Второй вложенный оператор оператора `if`, если он есть, достижим, если достижим сам оператор `if` и логическое выражение не является константным значением `true`.

Конечная точка оператора `if` достижима, если хотя бы один из его вложенных операторов достижим. Для оператора `if` без `else`-части конечная точка достижима, если достижим сам оператор `if`, а значение логического выражения не является константным значением `true`.

8.7.2. Оператор switch

Оператор `switch` позволяет выбрать для выполнения список операторов, связанный со switch-меткой, соответствующей значению выражения.

```
оператор-switch:
   switch  (  выражение  )  switch-блок

switch-блок:
   {  switch-секции_opt  }

switch-секции:
   switch-секция
   switch-секции   switch-секция

switch-секция:
   switch-метки   список-операторов

switch-метки:
   switch-метка
   switch-метки   switch-метка

switch-метка:
   case   константное-выражение   :
   default   :
```

Оператор-switch состоит из ключевого слова `switch`, за которым следует выражение, заключенное в круглые скобки, и *switch-блока*. *Switch-блок* состоит из нуля

или более заключенных в фигурные скобки *switch-секций*. Каждая *switch-секция* состоит из одной или нескольких *switch-меток*, за которыми следует *список-операторов* (раздел 8.2.1).

Управляющий тип оператора `switch` определяется по типу выражения.

- Если тип switch-выражения `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `bool`, `char`, `string`, *перечисление* или соответствующий им обнуляемый тип, то этот тип считается управляющим типом оператора `switch`.
- Иначе должно существовать ровно одно определенное пользователем неявное приведение (раздел 6.4) из типа switch-выражения к одному из возможных управляющих типов: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string` или соответствующему обнуляемому типу.
- Иначе, если не существует ни одного определенного пользователем неявного приведения или таких приведений несколько, выдается ошибка компиляции.

Константное выражение в каждой метке `case` должно неявно приводиться к управляющему типу оператора `switch`. Если две или более `case` метки в одном операторе `switch` имеют одинаковое значение, выдается ошибка компиляции.

В операторе `switch` не может быть больше одной метки `default`.

Оператор `switch` выполняется следующим образом:

- Вычисляется значение выражения и преобразуется к управляющему типу.
- Если одно из константных значений у одной из меток `case` равно значению вычисленного выражения, управление передается на список операторов, следующих за этой меткой `case`.
- Если значение выражения не равно ни одному из константных значений в метках `case` этого оператора, а метка `default` есть, то управление передается на список операторов, следующих за меткой `default`.
- Если значение выражения не равно ни одному из константных значений в метках `case` этого оператора, а метка `default` отсутствует, то управление передается на конечную точку оператора `switch`.

Если конечная точка в списке операторов любой из секций `switch` в операторе достижима, выдается ошибка компиляции. Это правило известно как «no fall through», оно не позволяет выполняться нескольким секциям подряд.

Пример:

```
switch (i)
{
   case 0:
      CaseZero();
      break;
   case 1:
      CaseOne();
      break;
   default:
      CaseOthers();
      break;
}
```

допустим, потому что ни одна из `switch` секций не имеет достижимой конечной точки. В отличие от языков C и C++, после выполнения секции не допускается выполнение следующей секции. Пример:

```
switch (i)
{
   case 0:
      CaseZero();
   case 1:
      CaseZeroOrOne();
   default:
      CaseAny();
}
```

даст ошибку компиляции. Если необходимо после выполнения одной `switch` секции перейти к какой-либо другой, следует использовать операторы перехода `goto case` или `goto default`:

```
switch (i)
{
   case 0:
      CaseZero();
      goto case 1;
   case 1:
      CaseZeroOrOne();
      goto default;
   default:
      CaseAny();
      break;
}
```

Допускается использовать несколько меток для одной `switch` секции. Например:

```
switch (i)
{
   case 0:
      CaseZero();
      break;
   case 1:
      CaseOne();
      break;
   case 2:
   default:
      CaseTwo();
      break;
}
```

Этот пример не противоречит правилам языка, потому что метки `case 2:` и `default:` относятся к одной и той же `switch` секции.

Правило недостижимости конечной точки `switch` секций предотвращает распространенную ошибку при использовании языков C и C++, когда `break` пропускают случайно. К тому же, это правило позволяет изменять порядок секций `switch` внутри одного оператора без каких-либо последствий. Например, в приведенном выше операторе `switch` можно переставить секции `switch` в обратном порядке:

```
switch (i)
{
   default:
      CaseAny();
      break;
   case 1:
      CaseZeroOrOne();
      goto default;
   case 0:
      CaseZero();
      goto case 1;
}
```

Список операторов в `switch` секции обычно заканчивается оператором `break`, `goto case` или `goto default`, но и другие способы обеспечить недостижимость конечной точки `switch` секции тоже допустимы. Например, оператор `while` с условием в виде логической константы `true` никогда не достигает конечной точки этого оператора. Операторы `throw` и `return` всегда передают управление в другое место и никогда не достигают своей конечной точки. Таким образом, следующий пример является допустимым:

```
switch (i)
{
   case 0:
      while (true) F();
   case 1:
      throw new ArgumentException();
   case 2:
      return;
}
```

ДЖОН СКИТ

Не думаю, что для создателей языка C# было разумным перетряхнуть все правила оператора `switch/case`. Область видимости объявленных внутри него переменных весьма неожиданна, использование оператора `break` мне также кажется ошибочным. Полагаю, разработчики думают о `switch` секциях как о блоках, так почему было не воплотить эти мысли?

```
case 0:
{
// Далее следует код для case 0 без необходимости использовать break
}
```

Аналогично, почему бы для нескольких `case` в одной секции не перечислить выражения через запятую, чтобы не повторять несколько меток `case` целиком?

Управляющим типом в операторе `switch` может быть `string`. Например:

```
void DoCommand(string command)
{
   switch (command.ToLower())
   {
      case "run":
         DoRun();
         break;
```

продолжение ⇨

```
      case "save":
         DoSave();
         break;
      case "quit":
         DoQuit();
         break;
      default:
         InvalidCommand(command);
         break;
   }
}
```

Подобно операции сравнения строк (раздел 7.10.7), выполнение оператора `switch` чувствительно к регистру. Нужная `switch` секция будет выполнена, только если значение выражения в точности равно константе в метке `case`.

Когда управляющим типом оператора `switch` является `string`, в метке `case` допускается использовать `null`.

Списки-операторов switch блока могут содержать объявления (раздел 8.5). Областью видимости локальной переменной или константы является `switch` блок.

> **БИЛ ВАГНЕР**
>
> Полезно представить, что каждый блок `switch` как бы заключен в невидимые фигурные скобки.

В пределах `switch` блока значение идентификатора, используемое в контексте выражения, должно быть одним и тем же (раздел 7.6.2.1).

Список операторов секции `switch` считается доступным, если доступен сам оператор `switch` и верно по крайней мере одно из утверждений:

- Выражение в операторе `switch` не является константой.
- Выражение в операторе `switch` является константой, которая равна константе в метке одной из секций `switch`.
- Выражение в операторе является константой, которая не равна ни одной константе в метках `switch` секций, но есть секция с меткой `default`.
- На метку в секции `switch` ссылается достижимый оператор `goto case` или `goto default`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Это правило не распространяется на случай, когда `goto case` или `goto default` находятся внутри блоков `try` или `catch`, оператор `try` содержит блок `finally`, конечная точка которого недостижима, а метка в секции `switch` находится вне этого оператора `try`.

Конечная точка оператора `switch` достижима, если хотя бы одно из следующих утверждений верно:

- Оператор `switch` содержит достижимый оператор `break`, который позволяет выйти из `switch` оператора.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Это правило не распространяется на случай, когда `break` находятся внутри блоков `try` или `catch`, оператор `try` содержит блок `finally`, конечная точка которого недостижима, а место назначения `break` находится вне этого оператора `try`.

- Оператор `switch` достижим, выражение в нем не является константой и в операторе нет метки `default`.
- Оператор `switch` достижим, выражение в нем является константой, которая не совпадает ни с одной константой в метке `case`, и нет метки `default`.

8.8. Операторы цикла

Операторы цикла многократно выполняют вложенный оператор.

оператор-цикла:
  оператор-while
  оператор-do
  оператор-for
  оператор-foreach

8.8.1. Оператор while

Оператор `while` в зависимости от условия выполняет вложенный оператор ноль или более раз.

оператор-while
  while **(** *логическое-выражение* **)** *вложенный-оператор*

Оператор `while` выполняется следующим образом:

- Вычисляется *логическое-выражение* (раздел 7.20).
- Если результат логического выражения равен `true`, управление передается вложенному оператору. Когда управление доходит до конечной точки вложенного оператора (возможно, с помощью `continue`), управление возвращается к началу оператора `while`.
- Если логическое выражение имеет значение `false`, управление передается на конечную точку оператора `while`.

Внутри вложенного оператора можно использовать оператор `break` (раздел 8.9.1) для передачи управления в конечную точку оператора `while`, таким образом заканчивая повторение вложенного оператора. Оператор `continue` (раздел 8.9.2) используется для передачи управления в конечную точку вложенного оператора, таким образом переходя к следующей итерации цикла `while`.

Вложенный оператор достижим, если оператор `while` достижим и логическое выражение не равно константе `false`.

Конечная точка оператора `while` достижима, если хотя бы одно из следующих утверждений верно:

- В операторе `while` есть достижимый оператор `break`, с помощью которого происходит выход из оператора `while`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Это правило не распространяется на случай, когда `break` находятся внутри блоков `try` или `catch`, оператор `try` содержит блок `finally`, конечная точка которого недостижима, а место назначения `break` находится вне этого оператора `try`.

- Оператор `while` достижим и логическое выражение не равно константе `true`.

8.8.2. Оператор do

Оператор `do` в зависимости от условия, которое он содержит, выполняет вложенный оператор один или несколько раз.

оператор-do:
  do *вложенный-оператор* **while** **(** *логическое-выражение* **)** **;**

Оператор `do` выполняется следующим образом:

- Управление передается вложенному оператору.
- Когда и если управление достигает конечной точки вложенного оператора (возможно, с помощью оператора `continue`), вычисляется *логическое-выражение* (раздел 7.20). Если логическое выражение имеет значение `true`, управление передается на начало оператора `do`. Иначе управление передается на конечную точку оператора `do`.

Внутри вложенного оператора можно использовать оператор `break` (раздел 8.9.1) для передачи управления на конечную точку оператора `do`, таким образом заканчивая повторение вложенного оператора. Оператор `continue` (раздел 8.9.2) используется для передачи управления в конечную точку вложенного оператора, таким образом переходя к следующей итерации цикла `do`.

Вложенный оператор оператора `do` достижим, если достижим сам оператор `do`.

Конечная точка оператора `do` достижима, если хотя бы одно из следующих условий верно:

- Оператор `do` содержит достижимый оператор `break`, который выполняет выход из этого оператора.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Это правило не распространяется на случай, когда `break` находятся внутри блоков `try` или `catch`, оператор `try` содержит блок `finally`, конечная точка которого недостижима, а место назначения `break` находится вне этого оператора `try`.

- Конечная точка вложенного оператора достижима и логическое выражение не равно константе `true`.

8.8.3. Оператор for

Оператор `for` вычисляет последовательность инициализирующих выражений, а затем, пока условие равно `true`, многократно выполняет вложенный оператор и вычисляет последовательность выражений итерации.

оператор-for
  for (*инициализатор-for*$_{opt}$; *условие-for*$_{opt}$; *итератор-for*$_{opt}$)
    вложенный-оператор

инициализатор-for:
  объявление-локальной-переменной
  список-операторов-выражений

условие-for:
  логическое-выражение

итератор-for:
  список-операторов-выражений

список-операторов-выражений
  оператор-выражение
  список-операторов-выражений , *оператор-выражение*

Инициализатор-for, если он есть, состоит либо из *объявления-локальной-переменной* (раздел 8.5.1), либо из *списка-выражений* (раздел 8.6), разделенных запятыми. Область видимости локальной переменной, объявленной в *инициализаторе-for*, начинается с *описателя-локальной-переменной* и распространяется до конца вложенного оператора, включая *условие-for* и *итератор-for*.

Условие-for, если оно есть, должно быть *логическим-выражением* (раздел 7.20).

Итератор-for, если он есть, состоит из списка *операторов-выражений* (раздел 8.6), разделенных запятыми.

Оператор `for` выполняется следующим образом:

- Если *инициализатор-for* присутствует, инициализаторы переменных и операторы-выражения выполняются в том порядке, в котором они написаны. Это шаг выполняется только один раз.
- Если *условие-for* присутствует, оно вычисляется.
- Если *условие-for* отсутствует или если его вычисление дает значение `true`, выполняется вложенный оператор. Когда и если управление доходит до конечной точки вложенного оператора (возможно, с помощью оператора `continue`), по порядку выполняются выражения *итератора-for*, если они есть. Затем выполняется следующая итерация, начиная с вычисления *условия-for* (см. предыдущий шаг).
- Если *условие-for* присутствует и его вычисление дает значение `false`, управление передается на конечную точку оператора `for`.

Внутри вложенного оператора можно использовать оператор `break` (раздел 8.9.1) для передачи управления на конечную точку оператора `for`, таким образом заканчивая повторение вложенного оператора. Оператор `continue` (раздел 8.9.2) используется для передачи управления на конечную точку вложенного опе-

ратора (таким образом выполняя *итератор-for* и переходя к следующей итерации цикла `for`, начиная с *условия-for*).

Вложенный оператор достижим, если одно из следующих условий верно:

- Оператор `for` достижим и он не имеет *условия-for*.
- Оператор `for` достижим, *условие-for* есть и оно не равно константе `false`.

Конечная точка оператора `for` достижима, если верно хотя бы одно из следующих утверждений:

- В операторе `for` есть достижимый оператор `break`, который выполняет выход из оператора `for`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Это правило не распространяется на случай, когда `break` находятся внутри блоков `try` или `catch`, оператор `try` содержит блок `finally`, конечная точка которого недостижима, а место назначения `break` находится вне этого оператора `try`.

- Оператор `for` достижим, *условие-for* в нем есть и оно не равно константе `true`.

8.8.4. Оператор foreach

Оператор `foreach` перебирает элементы в коллекции, выполняя вложенный оператор для каждого элемента.

```
оператор-foreach:
   foreach ( тип-локальной-переменной идентификатор in выражение )
      вложенный-оператор
```

Тип и *идентификатор* в операторе `foreach` объявляют **переменную цикла**. Если в качестве *типа локальной-переменной* используется `var` и в области видимости не определен тип с именем `var`, то переменная цикла имеет **неявный тип**, и ее фактический тип определяется по типу элементов в операторе `foreach` в соответствии с определенными ниже правилами. Переменная цикла представляет собой локальную переменную, доступную только для чтения, и ее область видимости распространяется на вложенный оператор. В процессе выполнения оператора `foreach` переменная цикла содержит элемент коллекции, для которого выполняется текущая итерация. При попытке изменить локальную переменную с помощью присваивания или операторов `++` и `--` выдается ошибка компиляции. Также ошибка компиляции выдается при попытке передать переменную цикла в качестве `ref` или `out` параметра.

> **КРИС СЕЛЛС**
>
> Использование оператора `foreach` облегчает чтение кода по сравнению с использованием оператора `for`.

ДЖОН СКИТ

Использование *одной* переменной цикла (которая доступна только для чтения и тем не менее волшебным образом изменяет свое значение на каждой итерации), вызывает распространенную ошибку, связанную с захваченными переменными. Код ниже выглядит так, как будто должен последовательно напечатать `"a", "b", "c", "d"`, но на самом деле он четыре раза печатает `"d"`.

```
List<Action> actions = new List<Action>();

foreach (string value in new[] { "a", "b", "c", "d" })
{
   actions.Add(() => Console.WriteLine(value));
}
foreach (Action action in actions)
{
   action();
}
```

Решение этой проблемы обычно предполагает объявление дополнительной переменной внутри тела оператора `foreach`, которой присваивается копия текущего значения переменной цикла. Таким образом, каждый делегат будет захватывать свою переменную. Первый цикл в таком случае будет выглядеть так:

```
foreach (string value in new[] { "a", "b", "c", "d" })
{
   String copy = value;
   actions.Add(() => Console.WriteLine(copy));
}
```

ПИТЕР СЕСТОФТ

Оператор `foreach` мог быть реализован по-другому и избежать той проблемы с переменной, о которой написал Джон. Для этого просто необходимо было перенести объявление переменной `V v` внутрь цикла `while` в конструкции `try-while-finally`, о которой будет сказано ниже. Более того, в первоначальном варианте спецификации языка C# объявление переменной было внутри цикла. Но в ходе стандартизации C# версии 2.0 объявление переменной `v` вне цикла было признано более близким к правде, по крайней мере, в реализации языка C# от Microsoft, и поэтому спецификация языка была изменена в соответствии с этим.

При компиляции для оператора `foreach` сначала определяется **тип коллекции**, **тип перечислителя** и **тип элемента** выражения. Это происходит таким образом:

- Если тип *выражения* `X` — массив, то существует неявное ссылочное приведение из `X` к интерфейсу `System.Collections.IEnumerable`, так как `System.Array` реализует этот интерфейс. Тогда **тип коллекции** — это интерфейс `System.Collections.IEnumerable`, **тип перечислителя** — интерфейс `System.Collections.IEnumerator`, а **тип элементов** — это тип элемента массива типа `X`.
- Если тип *выражения* `X` — `dynamic`, то существует неявное приведение из *выражения* к интерфейсу `System.Collections.IEnumerable` (раздел 6.1.8). Тогда **тип коллекции** — это интерфейс `System.Collections.IEnumerable`,

тип перечислителя — интерфейс `System.Collections.IEnumerator`. Если тип *переменной-цикла* задан как `var`, то **тип элементов** будет `dynamic`, в противном случае — `object`.

- Иначе определяется, поддерживает ли тип `X` соответствующий метод `GetEnumerator`:
  - Для типа `X` выполняется поиск элемента с названием `GetEnumerator` без аргументов типа. Если поиск не дал совпадения, возникла неоднозначность или результатом не является группа методов, происходит проверка на приведение к интерфейсу `IEnumerable`, как описано ниже. Если поиск методов дал результат, отличающийся от группы методов или отсутствия совпадения, рекомендуется выдать предупреждение.

> **ЭРИК ЛИППЕРТ**
>
> Этот подход, основанный на паттернах, как будто возвращает нас во времена, когда обобщенный интерфейс `IEnumerable<T>` еще был недоступен и разработчикам коллекций приходилось использовать более точные аннотации типов для перечислителей. Реализации необобщенного интерфейса `IEnumerable` всегда заканчиваются упаковкой каждого элемента коллекции целых чисел, потому что возвращаемый тип свойства `Current` — это `object`. Сервер, предоставляющий коллекции чисел, мог бы предоставить не основанные на интерфейсах `GetEnumerable`, `MoveNext` и `Current`, такие что свойство `Current` возвращает неупакованное значение.
>
> Естественно, для мира, в котором есть обобщенный интерфейс `IEnumerable<T>`, все эти трудности становятся ненужными. Подавляющее большинство коллекций реализует этот интерфейс.

  - Выполняется разрешение перегрузки с использованием полученной группы методов и пустого списка аргументов. Если разрешение перегрузки не нашло подходящих методов, обнаружило неоднозначность или единственный лучший метод оказался статическим или не `public`, происходит проверка на приведение к интерфейсу `IEnumerable`, как описано ниже. Если поиск методов дал результат, отличающийся от единственного `public` метода экземпляра или отсутствия совпадения, рекомендуется выдать предупреждение.
  - Если возвращаемый методом `GetEnumerator` тип `E` не является классом, структурой или интерфейсом, то выдается ошибка и следующие шаги не выполняются.
  - Выполняется поиск для типа `E` с идентификатором `Current` без аргументов типа. Если поиск не дал совпадения, закончился ошибкой или дал результат, отличающийся от `public` свойства экземпляра, доступного для чтения, выдается ошибка и дальнейшие шаги не выполняются.
  - Выполняется поиск для типа `E` с идентификатором `MoveNext` без аргументов типа. Если поиск не дал совпадения, закончился ошибкой или дал результат, отличающийся от группы методов, выдается ошибка и дальнейшие шаги не выполняются.

    - Выполняется разрешение перегрузки для полученной группы методов с пустым списком аргументов. Если разрешение перегрузки не нашло подходящих методов, обнаружило неоднозначность или единственный лучший метод оказался статическим, не `public` или его тип возврата не `bool`, выдается ошибка и дальнейшие шаги не выполняются.
    - **Тип коллекции** — `X`, **тип перечислителя** — `E`, **тип элемента** — тип свойства `Current`.
- В противном случае выполняется проверка на перечислимый интерфейс:
    - Если существует ровно один тип `T` такой, что существует неявное приведение из `X` к интерфейсу `Sistem.Collections.Generic.IEnunerable<T>`, то **типом коллекции** считается этот интерфейс, **типом перечислителя** будет интерфейс `Sistem.Collections.Generic.IEnunerable<T>`, а **типом элемента** будет `T`.
    - Иначе, если существует несколько таких типов `T`, то выдается ошибка и следующие шаги не выполняются.
    - Иначе, если существует неявное приведение из `T` к интерфейсу `System.Collections.IEnumerable`, то **типом коллекции** считается этот интерфейс, **тип перечислителя** — `System.Collections.IEnumerator`, а **типом элемента** будет `object`.
- Иначе выдается ошибка, и следующие шаги не выполняются.

ПИТЕР СЕСТОФТ

Прискорбно, но способ обработки компилятором оператора `foreach` означает, что он связан динамически. Например, если взять произвольный не бесплодный класс `C` и интерфейс `I`, то следующий код считается допустимым и не вызовет при компиляции ни ошибок, ни предупреждений:

```
C[] xs = ...;
foreach (I x in xs)
...
```

Совсем недавно я на этом попался, когда удалял ненужный, как мне показалось, интерфейс `I` из реализуемых классом `C` интерфейсов. Проект был построен успешно, но на этапе выполнения приложение завершилось исключением `InvalidCastException` из-за того, что я всего лишь пропустил несколько операторов `foreach`. На самом деле такое поведение вполне предсказуемо, потому что возможен какой-то наследник класса `C`, реализующий интерфейс `I`. И что еще более удивительно, в приведенном примере интерфейс `I` и класс `C` можно поменять местами, а суть этого примера не изменится.

Если все перечисленные шаги выполнены успешно, то однозначно определены тип коллекции `C`, тип перечислителя `E` и тип элемента `T`. Оператор `foreach` из вида

```
foreach(V v in x) вложенный-оператор
```

будет развернут к виду:

```
{
   E e = ((C)(x)).GetEnumerator();
   try
   {
      V v;
      while (e.MoveNext())
      {
         v = (V)(T)e.Current;
         вложенный-оператор
      }
   }
   finally
   {
   ... // Освобождение e
   }
}
```

Переменная `e` не видима и не доступна в выражении `x`, во вложенном операторе и любых других местах исходного кода программы. Переменная `v` во вложенном операторе доступна только для чтения. Если нет явного приведения (раздел 6.2) из типа элемента `T` к типу `V` (*типу локальной-переменной* оператора `foreach`), то выдается ошибка компиляции и следующие шаги не выполняются. Если `x` имеет значение `null`, то на этапе выполнения будет выброшено исключение `System.NullReferenceException`.

> **МАРЕК САФАР**
>
> Оператор `foreach` – это классический пример развития языка. C# 1.0 не имел обобщенного интерфейса `IEnumerable<T>` и в нем необходимо было использовать явное приведение из типа элемента к типу локальной переменной. В C# 2.0 был реализован обобщенный вариант оператора `foreach` с элементами обобщенного типа. C# 3.0 поставил эту конструкцию на то место, где она должна была находиться с самого начала, путем с реализацией возможности для переменной цикла использовать **неявный тип**, что изменяет явное приведение на неявное и позволяет избежать любых исключений `InvalidCastException` во время выполнения программы.

Разным реализациям языка C# разрешается обрабатывать оператор `foreach` по-разному,— например, с целью увеличения быстродействия, — но они в любом случае должны обеспечивать совместимость с описанным выше поведением.

Действия, выполняемые в блоке `finally`, зависят от следующих условий:

- Если есть неявное приведение из типа перечислителя к интерфейсу `SystemIDsposable`, тогда:
  - Если перечислитель имеет необнуляемый тип-значение, то блок `finally` расширяется в эквивалент кода:

    ```
    finally
    {
       ((System.IDisposable)e).Dispose();
    }
    ```

  - В противном случае блок `finally` расширяется в эквивалент кода:

```
finally
{
   if (e != null) ((System.IDisposable)e).Dispose();
}
```

но если `E` имеет тип-значение или параметр-тип, инстанцированный в тип-значение, то преобразование переменной `e` к интерфейсу `SystemIDsposable` не вызывает операцию упаковки.

- Иначе, если перечислитель имеет бесплодный тип, блок `finally` будет пустым:

```
finally
{
}
```

- В противном случае блок `finally` будет следующим:

```
finally
{
   System.IDisposable d = e as System.IDisposable;
   if (d != null) d.Dispose();
}
```

Локальная переменная `d` не видна и недоступна в любой точке кода пользователя. Она не конфликтует с любой другой переменной, область видимости которой содержит блок `finally`.

ДЖОН СКИТ

В C# 2.0 оператор `foreach` освобождает используемый перечислитель, что делает блоки итератора более удобными: вполне разумно получить ресурс для выполнения цикла, а затем удалить его, когда цикл закончится или будет выполнен выход из цикла.

Порядок, в котором `foreach` проходит по элементам, следующий. Для одномерных массивов элементы просматриваются в порядке возрастания индекса от `0` до `Length – 1`. Для многомерных массивов сначала увеличиваются индексы в самой правой размерности, затем – в следующей слева размерности и т. д.

Следующий пример выводит на экран каждое значение в двумерном массиве в порядке следования элементов:

```
using System;
class Test
{
   static void Main()
   {
      double[,] values =
      {
         {1.2, 2.3, 3.4, 4.5},
         {5.6, 6.7, 7.8, 8.9}
      };
      foreach (double elementValue in values)
         Console.Write("{0} ", elementValue);
      Console.WriteLine();
   }
}
```

Результат:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

В примере

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

тип переменной `n` выводится как `int` по типу элементов массива `numbers`.

8.9. Операторы перехода

Операторы перехода выполняют безусловную передачу управления.

```
оператор-перехода:
   оператор-break
   оператор-continue
   оператор-goto
   оператор-return
   оператор-throw
```

Место в коде, куда операторы перехода передают управление, называют **местом назначения** (target) оператора перехода.

Когда оператор перехода встречается внутри блока, а место назначения этого оператора вне блока, оператор выполняет **выход** из блока. Операторы перехода могут передавать управление из блока вовне, но никогда — внутрь блока.

Порядок выполнения оператора перехода усложняется наличием операторов `try`. Если операторы `try` отсутствуют, оператор перехода выполняет безусловный переход к своему месту назначения. При наличии оператора `try` выполнение оператора перехода более сложно. Если оператор перехода выполняет выход из одного или нескольких операторов `try`, имеющих блок `finally`, то сначала управление передается блоку `finally` самого внутреннего оператора `try`. Когда и если управление достигает конечной точки этого `finally` блока, управление передается блоку `finally` объемлющего оператора `try`. Этот процесс повторяется, пока не будут выполнены блоки `finally` всех операторов `try`.

В примере

```
using System;
class Test
{
   static void Main()
   {
      while (true)
      {
         try
         {
            try
            {
               Console.WriteLine("Перед break");
               break;
            }
```

```
            finally
            {
               Console.WriteLine("Вложенный finally блок");
            }
         }
         finally
         {
            Console.WriteLine("Внешний finally блок");
         }
      }
      Console.WriteLine("После break");
   }
}
```

прежде чем управление будет передано на место назначения оператора перехода, будут выполнены блоки `finally` двух операторов `try`.

Результат:

```
Before break
Innermost finally block
Outermost finally block
After break
```

8.9.1. Оператор break

Оператор `break` выполняет выход из ближайшего объемлющего оператора `switch`, `while`, `do`, `for` и `foreach`.

```
оператор-break:
   break   ;
```

Местом назначения оператора `break` является конечная точка ближайшего оператора `switch`, `while`, `do`, `for` или `foreach`, в котором он находится. Если оператор `break` не находится внутри этих операторов, выдается ошибка компиляции.

Если несколько операторов `switch`, `while`, `do`, `for` или `foreach` находятся внутри друг друга, то оператор `break` применяется только к самому внутреннему из них. Чтобы выйти сразу из нескольких вложенных операторов, необходимо использовать оператор `goto` (раздел 8.9.3).

Оператор `break` не может использоваться для выхода из блока `finally`, его место назначения должно располагаться внутри этого блока. В противном случае выдается ошибка компиляции.

Оператор `break` выполняется следующим образом:

- Если оператор `break` выполняет выход из одного или нескольких операторов `try`, имеющих блок `finally`, то сначала выполняется блок `finally` самого внутреннего оператора `try`. Если управление достигает конечной точки этого блока `finally`, то управление передается к блоку `finally` объемлющего оператора `try`. Этот процесс повторяется, пока не будут выполнены блоки `finally` всех операторов `try`.
- Управление передается в конечную точку оператора `break`.

Так как оператор `break` всегда выполняет безусловный переход, его конечная точка недостижима.

> **ДЖЕССИ ЛИБЕРТИ**
>
> Всякий раз, когда вы видите оператор безусловного перехода, а особенно — выход из метода, рассматривайте его как ужасную проблему (то есть с криками выбегайте из комнаты). Почти каждый случай использования операторов `break`, `continue` и `goto` можно переписать или выполнить рефакторинг без их использования. В итоге код, скорее всего, получится более простым, понятным и поддерживаемым.

8.9.2. Оператор continue

Оператор `continue` начинает новую итерацию ближайшего объемлющего оператора `while`, `do`, `for` и `foreach`.

```
оператор-continue:
  continue  ;
```

Местом назначения `continue` является конечная точка вложенного оператора ближайшего объемлющего оператора `while`, `do`, `for` и `foreach`. Если оператор `continue` не находится внутри вышеуказанных операторов, выдается ошибка компиляции.

Если несколько операторов `switch`, `while`, `do`, `for`, `foreach` находятся внутри друг друга, то оператор `continue` применяется только к самому внутреннему из них. Чтобы выйти сразу из нескольких вложенных перечисленных операторов, нужно использовать оператор `goto` (раздел 8.9.3).

Оператор `continue` не может использоваться для выхода из блока `finally`, его место назначения должно располагаться внутри этого блока. В противном случае выдается ошибка компиляции.

Оператор `continue` выполняется следующим образом:

- Если оператор `continue` выполнит выход из одного или нескольких операторов `try`, имеющих блок `finally`, то сначала выполняется блок `finally` самого внутреннего оператора `try`. Если управление достигает конечной точки этого блока `finally`, то управление передается к блоку `finally` объемлющего оператора `try`. Этот процесс повторяется, пока не будут выполнены блоки `finally` всех операторов `try`.
- Управление передается в конечную точку оператора `continue`.

Так как оператор `continue` всегда выполняет безусловный переход, его конечная точка недостижима.

> **ПЕТЕР СЕСТОФТ**
>
> Некоторые языки программирования, в том числе Java, имеют операторы `break` и `continue`, для которых указывается, какой из вложенных циклов нужно продолжить или завершить. Тот же самый эффект в C# достигается использованием оператора `goto`. Хотя операторы `break` и `continue` представляют собой более упорядоченную версию `goto`, за ними почти так же трудно уследить. По моему мнению, они ухудшают читабельность даже небольших программ. Смотрите примеры номер 78 и 79 в моей книге «Java Precisely», второе издание.

8.9.3. Оператор goto

Оператор `goto` передает управление оператору, который помечен указанной меткой.

оператор-goto:
  goto *идентификатор* **;**
  goto case *константное-выражение* **;**
  goto default **;**

Местом назначения оператора `goto` *идентификатор* является оператор, помеченный указанной меткой. Если метки с этим именем в текущем функциональном элементе нет или оператор `goto` находится вне зоны видимости метки, выдается ошибка компиляции. Это правило позволяет использовать оператор `goto` для *выхода* из вложенной области, но не для передачи управления *внутрь* нее.

В примере

```
using System;
class Test
{
   static void Main(string[] args)
   {
      string[,] table =
      {
         {"Red", "Blue", "Green"},
         {"Monday", "Wednesday", "Friday"}
      };

      foreach (string str in args)
      {
         int row, colm;
         for (row = 0; row <= 1; ++row)
            for (colm = 0; colm <= 2; ++colm)
               if (str == table[row,colm])
                  goto done;
               Console.WriteLine("{0} not found", str);
               continue;
         done:
         Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
      }
   }
}
```

оператор `goto` используется для передачи управления из внутреннего блока.

Местом назначения оператора `goto case` является список операторов в непосредственно объемлющем его операторе `switch` (раздел 8.7.2), который содержит метку с соответствующим константным значением. Если `goto case` не находится внутри оператора `switch`, если *константное-выражение* не может быть неявно приведено (раздел 6.1) к управляющему типу оператора `switch` или если `switch` не содержит метки `case` с данным константным значением, выдается ошибка компиляции.

Местом назначения оператора `goto default` является список операторов в непосредственно объемлющем его операторе `switch` (раздел 8.7.2), который содержит метку `default`. Если `goto default` не находится в пределах оператора `switch` либо если он не содержит метки `default`, выдается ошибка компиляции.

Оператор `goto` не может выполнять выход из блока `finally` (раздел 8.10). Если `goto` встречается внутри блока `finally`, то его место назначения должно находиться в пределах этого блока `finally`, иначе выдается ошибка компиляции.

> **БИЛ ВАГНЕР**
> Это правило делает оператор `goto` чуть менее ужасным для C#, чем чистое зло, но я пока не нашел, где его можно применить.

Оператор `goto` выполняется следующим образом:

- Если оператор `goto` выполняет выход из одного или нескольких операторов `try`, имеющих блок `finally`, то сначала выполняется блок `finally` самого внутреннего оператора `try`. Если управление достигает конечной точки этого блока `finally`, то управление передается к блоку `finally` объемлющего оператора `try`. Этот процесс повторяется, пока не будут выполнены блоки `finally` всех операторов `try`.
- Управление передается в место назначения оператора `goto`.

Так как оператор `goto` выполняет безусловный переход, его конечная точка недостижима.

> **КРИС СЕЛЛС**
> Пожалуйста, не используйте у себя в коде метки и оператор `goto`. Я еще не видел кода, который не выиграл бы от их отсутствия.

> **КРИСТИАН НЕЙГЕЛ**
> Оператор `goto` имеет смысл использовать внутри `switch` для обработки нескольких секций подряд, в других случаях его применять не следует.

> **ПИТЕР СЕСТОФТ**
> В статье 1974 года «Структурное программирование с операторами goto» Дон Кнут осветил многие альтернативные (в сравнении с `goto`) приемы программирования. Он показал целесообразность использования оператора `goto`. В конце своей статьи Кнут сделал вывод, что «нам в самом деле следует запретить `goto`... хотя бы для того, чтобы вынудить людей более тщательно формулировать свои абстракции». Даже просто список благодарностей этой статьи показывает, «кто есть кто» среди родоначальников программирования.
> Самая веская причина включения оператора `goto` в C# состоит в том, что вам может потребоваться *генерировать* код, который использует `goto` — например, для таких программ, как синтаксические или лексические анализаторы, автоматы, диаграммы состояний, виртуальные машины и аналогичных.

8.9.4. Оператор return

Оператор `return` возвращает управление в точку вызова функционального элемента, внутри которого он находится.

оператор-return:
  return *выражение*$_{opt}$ **;**

Оператор `return` без выражения можно использовать только внутри функциональных элементов, которые не возвращают значение, то есть метода, возвращающего `void`, кода доступа `set` свойства или индексатора, кодов доступа `add` или `remove` события, конструктора экземпляра, статического конструктора и деструктора.

Оператор `return` с выражением можно использовать только внутри функциональных элементов, которые возвращают значение, то есть метода, возвращающего значение, отличное от `void`, кода доступа `get` свойства или индексатора и определенного пользователем оператора. Должно существовать неявное приведение (раздел 6.1) из типа выражения к типу, возвращаемому функциональным элементом.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Если оператор `return` находится внутри анонимной функции, вместо этих правил применяются правила из раздела 6.5.

Если оператор `return` встречается внутри блока `finally`, выдается ошибка компиляции.

Оператор `return` выполняется следующим образом:

- Если оператор `return` содержит выражение, то оно вычисляется и результат преобразуется с помощью неявного приведения к типу, возвращаемому методом. Результат приведения становится значением, которое возвращается вызывающему коду.
- Если оператор `return` находится внутри одного или нескольких операторов `try`, имеющих блок `finally`, то сначала выполняется блок `finally` самого внутреннего оператора `try`. Если управление достигает конечной точки этого блока `finally`, то управление передается к блоку `finally` объемлющего оператора `try`. Этот процесс повторяется, пока не будут выполнены блоки `finally` всех операторов `try`.
- Управление возвращается в точку вызова данного функционального элемента.

Оператор `return` безусловно возвращает управление, и, следовательно, его конечная точка недостижима.

8.9.5. Оператор throw

Оператор `throw` выбрасывает исключение.

оператор-throw:
  throw *выражение*$_{opt}$ **;**

Оператор `throw` с выражением выбрасывает значение, получаемое путем вычисления выражения. Выражение должно задавать значение типа `System.Exception`,

наследника от `System.Exception` либо типа, тип параметра которого является `System.Exception` или его наследником в качестве эффективного базового класса. Если вычисление выражения дает `null`, выбрасывается исключение `System.NullReferenceException`.

> **ПИТЕР СЕСТОФТ**
>
> По этой причине исключения, которые генерируются в C#, не могут быть `null`, и выбор типа исключения в конструкция `try-catch` по его классу имеет смысл. На самом деле команда промежуточного языка .NET/CLI `throw` ведет себя в этом смысле так же, как оператор `throw` языка C#. Таким образом, даже если исключение сгенерировано на другом языке .NET/CLI, оно все равно будет не `null` при обработке оператором `try-catch`.

Оператор `throw` без выражения можно использовать только внутри блока `catch`. В таком случае он заново выбрасывает исключение, которое было перехвачено в блоке `catch`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Оператор `throw` без выражения не допускается использовать в блоке `finally`, а также в анонимных функциях непосредственно в блоке `catch`.

```
delegate void F();
class Program
{
   static void Main()
   {
      try
      {
      }
      catch
      {
         F f = () => { throw; }; // Ошибка CS0156
         try
         {
         }
         finally
         {
            throw; // Ошибка CS0724
         }
      }
   }
}
```

Оператор `throw` безусловно передает управление, а значит, его конечная точка недостижима.

Когда генерируется исключение, то управление передается блоку `catch` самого внутреннего из операторов `try`, которые могут это исключение перехватить. Процесс, начинающийся от места, где исключение было создано, до точки, где оно было перехвачено, называется **распространением исключения**. Распространение

исключения состоит из повторяющихся шагов, пока не будет найден блок `catch`, перехватывающий это исключение. В этом описании точкой **генерации исключения** является место, в котором выбрасывается исключение.

- В текущем функциональном элементе проверяются все операторы `try`, внутри которых находится точка генерации исключения. Для каждого из этих операторов `try`, начиная с самого внутреннего и заканчивая самым внешним, выполняются следующие проверки:
  - Если `try`-блок `S` содержит точку генерации исключения и если `S` имеет один или более блоков `catch`, то ищется подходящий блок `catch` в порядке их следования в коде. Первый блок `catch`, который определяет тип этого исключения или базовый для него, считается подходящим. Общий блок `catch` (раздел 8.10) считается подходящим для исключения любого типа. Если подходящий блок найден, распространение исключения заканчивается и управление передается этому блоку `catch`.
  - В противном случае, если `try` блок или `catch` блок `S` содержит точку генерации исключения и если `S` имеет блок `finally`, управление передается блоку `finally`. Если внутри блока `finally` выбрасывается другое исключение, то процесс обработки текущего исключения прерывается. Иначе, если управление достигает конечной точки блока `finally`, процесс обработки текущего исключения продолжается.
- Если обработчик исключения не был найден внутри текущего вызова функционального элемента, его выполнение прерывается. Приведенная выше последовательность обработки исключения повторяется для вызывающего кода с точкой генерации исключения, соответствующей оператору, в котором был вызван функциональный элемент.
- Если обработка исключения привела к завершению вызова всех функциональных элементов в текущем потоке, что свидетельствует о том, что поток не имеет обработчика данного исключения, то выполнение потока завершается. Результат такого завершения потока определяется конкретной реализацией языка.

> **БИЛЛ ВАГНЕР**
>
> Этот процесс предполагает, что вам следует предусматривать базовую очистку в самых верхних методах всех ваших потоков. Иначе поведение вашего приложения будет неопределенным из-за исключений, которые могут привести к завершению потоков.

8.10. Оператор try

Оператор `try` обеспечивает способ перехвата исключений, которые произошли в процессе выполнения блока. Кроме того, оператор `try` обеспечивает возможность задать блок кода, который обязательно выполнится, когда управление покидает оператор `try`.

```
оператор-try:
   try  блок  catch-блоки
   try  блок  finally-блок
   try  блок  catch-блоки  finally-блок

catch-блоки:
   специальные-catch-блоки  общий-catch-блок_opt
   специальные-catch-блоки_opt  общий-catch-блок

специальные-catch-блоки
   специальный-catch-блок
   специальные-catch-блоки  специальный-catch-блок

специальный-catch-блок:
   catch  (  тип-класса  идентификатор_opt  )  блок

общий-catch-блок:
   catch  блок

finally-блок:
   finally  блок
```

Существует три формы оператора `try`:

- блок `try`, за которым следует один или несколько блоков `catch`.
- блок `try`, за которым следует блок `finally`.
- блок `try`, за которым следует один или несколько блоков `catch`, в конце следует блок `finally`.

Когда в блоке `catch` задан *тип-класса*, он должен быть типа `System.Exception`, наследника от `System.Exception`, либо типа, тип параметра которого является `System.Exception`, или его наследником в качестве эффективного базового класса.

Когда в блоке `catch` задан *тип-класса* и *идентификатор*, то создается **переменная исключения** с этим именем и типом. Эта переменная соответствует локальной переменной, область действия которой распространяется на весь блок `catch`. Во время выполнения блока `catch` переменная исключения представляет обрабатываемое исключение. Для целей проверки на явное присваивание переменная исключения считается явно присвоенной в пределах всего блока `catch`.

Если блок `catch` не содержит идентификатора, получить доступ к исключению в этом блоке `catch` невозможно.

Блок `catch`, в котором не определены ни идентификатор, ни тип, называется общим блоком `catch`. Оператор `try` может иметь только один общий блок `catch`, и если он есть, он должен располагаться после всех остальных блоков `catch`.

Некоторые языки программирования могут поддерживать исключения, которые нельзя представить как объект-наследник `System.Exception`, но такие исключения не могут быть сгенерированы кодом на C#. Такие исключения можно перехватить с помощью общего блока `catch`. Таким образом, общий блок `catch` семантически отличается от блоков, в которых задан тип `System.Exception`, и он может перехватывать исключения, созданные в других языках программирования.

ЭРИК ЛИППЕРТ

В текущей реализации языка C# и CLR Microsoft созданное исключение, не унаследованное от `System.Exception`, преобразуется к объекту `RutimeWrappedException`. И как следствие этого, блок `catch(Exception e)` перехватывает все исключения. Если вы хотите отключить эту возможность и обеспечить семантику C# 1.0, где исключения, не относящиеся к `Exception`, таким способом не перехватывались, используйте следующий атрибут сборки:

```
[assembly:System.Runtime.CompilerServices.RuntimeCompatibility(WrapNonException
    Throws = false)]
```

При обработке исключения блоки `catch` проверяются в лексическом порядке. Если какой-либо блок `catch` определен с тем же или родительским типом, который уже встречался в одном из предыдущих блоков `catch` этого оператора `try`, выдается ошибка компиляции. Это ограничение не допускает наличия недостижимых блоков `catch`.

Внутри блока `catch` можно использовать оператор `throw` (раздел 8.9.5) для повторной генерации исключения, которое было перехвачено данным блоком `catch`. Присваивание переменной блока `catch` других значений не изменяет исключение, которое генерируется повторно.

В примере

```
using System;
class Test
{

   static void F()
   {
      try
      {
         G();
      }
      catch (Exception e)
      {
         Console.WriteLine("Exception in F: " + e.Message);
         e = new Exception("F");
         throw;     // Повторная генерация исключения
      }
   }

   static void G()
   {
      throw new Exception("G");
   }

   static void Main()
   {
      try
      {
         F();
      }
      catch (Exception e)
```

продолжение ☞

```
        {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

в методе `F` перехватывается исключение, выводится некая диагностическая информация на консоль, изменяется переменная, после чего исключение генерируется снова. Повторно выброшенное исключение такое же, как и первоначальное, следовательно, вывод будет таким:

```
Exception in F: G
Exception in Main: G
```

Если бы первый блок `catch` выбрасывал исключение `e` вместо повторной генерации текущего исключения, вывод был бы таким:

```
Exception in F: G
Exception in Main: F
```

Если какой-либо из операторов `break`, `continue` или `goto` передает управление из блока `finally`, выдается ошибка компиляции. Если операторы `break`, `continue` или `goto` встречаются внутри блока `finally`, место их назначения должно находиться внутри блока `finally`.

Если внутри блока `finally` встречается оператор `return`, выдается ошибка компиляции.

Оператор `try` выполняется следующим образом:

- Управление передается блоку `try`.
- Когда и если управление достигает конечной точки блока `try`:
  - Если оператор `try` имеет блок `finally`, то он выполняется.
  - Управление передается в конечную точку оператора `try`.
- Если исключение распространяется на оператор `try` при выполнении блока `try`, то:
  - Блоки `catch`, если они есть, проверяются в порядке их следования для поиска подходящего обработчика исключения. Первый блок `catch`, который определяет тип этого исключения или базовый для него, считается подходящим. Общий блок `catch` считается подходящим для исключения любого типа. Если подходящий блок `catch` найден:
    - Если в подходящем блоке `catch` описана переменная исключения, ей присваивается это исключение.
    - Управление передается подходящему блоку `catch`.
    - Когда и если управление достигает конечной точки блока `catch`:
      — если оператор `try` имеет блок `finally`, он выполняется;
      — управление передается в конечную точку оператора `try`.
    - Если исключение распространяется на оператор `try` при выполнении блока `catch`, то:
      — если оператор `try` имеет блок `finally`, он выполняется;
      — исключение распространяется на следующий объемлющий оператор `try`.

- Если оператор `try` не имеет блоков `catch` или ни один из блоков `catch` не является подходящим:
  - если оператор `try` имеет блок `finally`, он выполняется;
  - исключение распространяется на следующий объемлющий оператор `try`.

ЭРИК ЛИППЕРТ

Если стек вызова включает в себя код, защищенный блоками `try-catch`, написанными на других языках (например, на Visual Basic), среда выполнения может использовать «фильтр исключений» для определения подходящего блока `catch`. Как следствие, пользовательский код может выполняться после генерации исключения, но перед соответствующим блоком `finally`. Если ваш код обработки исключений зависит от глобального состояния, которое поддерживается блоком `finally` до выполнения любого другого кода пользователя, вам следует принимать соответствующие меры, чтобы быть уверенным, что ваш код перехватит исключение прежде, чем среда выполнения выполнит заданные пользователем «фильтры исключений», которые могут быть в стеке.

Операторы блока `finally` выполняются всегда, когда управление покидает оператор `try`. Это происходит, если оператор `try` завершается нормально, если выход выполняется с помощью операторов `break`, `continue`, `goto` или `return` или как результат распространения исключения из оператора `try`.

Если исключение происходит внутри блока `finally` и не перехватывается внутри него, оно распространяется до следующего объемлющего оператора `try`. Если при этом происходил процесс распространения другого исключения, оно теряется. Процесс распространения исключения обсуждался при описании оператора `throw` (раздел 8.9.5).

БИЛЛ ВАГНЕР

Это поведение показывает, что особенно важно делать код блока `finally` таким, чтобы избежать возникновения нового исключения.

ДЖОН СКИТ

Было бы удобно *не делать* блок `finally` защищенным от исключений. Концепция ошибки, проистекающей от другой, в .Net framework уже реализована в виде «inner exception» («внутреннего исключения»). Случай, когда два исключения связаны, но не известно, какое из них является причиной, а какое следствием, на данный момент не обрабатывается. Другая больная тема — какие исключения может порождать данный метод. Java пытается решить эти проблемы с помощью «checked exception», но, на мой взгляд, по большей части неудачно.

Иногда я думаю, что мы (представители IT-технологий) достаточно хорошо делаем «успешные» сценарии, но в плане обработки ошибок нам еще очень много нужно сделать.

Блок `try` в операторе `try` достижим, если достижим сам оператор.
Блок `catch` в операторе `try` достижим, если достижим сам оператор.
Блок `finally` в операторе `try` достижим, если достижим сам оператор.
Конечная точка оператора `try` достижима, если оба утверждения верны:

- Достижима конечная точка блока `try` или конечная точка как минимум одного блока `catch`.
- Если блок `finally` присутствует, то его конечная точка достижима.

8.11. Операторы checked и unchecked

Операторы `checked` и `unchecked` используются для управления **контекстом проверки переполнения** при целочисленных арифметических операциях и преобразованиях типа.

оператор-checked:
  checked *блок*

оператор-unchecked:
  unchecked *блок*

Оператор `checked` заставляет вычислять все выражения в *блоке* в проверяемом контексте. Оператор `unchecked` заставляет вычислять все выражения в *блоке* в непроверяемом контексте.

Операторы `checked` и `unchecked` идентичны операциям `checked` и `unchecked` (раздел 7.6.12), но они действуют на блок, а не на одно выражение.

8.12. Оператор lock

Оператор `lock` получает для данного объекта взаимоисключающую блокировку, выполняет оператор и затем освобождает блокировку.

оператор-lock
  lock **(** *выражение* **)** *вложенный-оператор*

Выражение в операторе `lock` должно определять значение *ссылочного-типа*. Для типа этого выражения не выполняется неявная упаковка (раздел 6.1.7), и поэтому если результат вычисления выражения будет *типом-значением*, выдается ошибка компиляции.

Оператор `lock` вида

```
lock (x)
```

где `x` – выражение *ссылочного-типа*, в точности эквивалентен следующему коду:

```
bool __lockWasTaken = false;
try
{
   System.Threading.Monitor.Enter(x, __lockWasTaken);
```

```
...
}
finally
{
   if (__lockWasTaken)System.Threading.Monitor.Exit(x);
}
```

кроме того, что в операторе `lock` выражение `x` вычисляется только один раз.

> **БИЛЛ ВАГНЕР**
>
> Оператор `lock()` выполняет дополнительную проверку времени компиляции против блокирования типа-значения.

Пока блокировка установлена, код этого потока может также получать и снимать блокировку. Однако код других потоков не может получить блокировку, пока она не будет освобождена.

Установка блокировки объекта типа `System.Type` для синхронизации доступа к статическим данным не рекомендуется. Другой код может заблокировать тот же тип, что может привести к тупиковой ситуации взаимной блокировки. Более правильно для синхронизации доступа к статическим данным использовать блокировку статического `private` объекта. Например:

```
class Cache
{
   private static readonly object synchronizationObject = new object();
   public static void Add(object x)
   {
      lock (Cache.synchronizationObject)
      {
         ...
      }
   }

   public static void Remove(object x)
   {
      lock (Cache.synchronizationObject)
      {
         ...
      }
   }
}
```

> **ДЖОЗЕФ АЛЬБАХАРИ**
>
> Хороший подход при написании библиотек общего пользования — это делать статические методы безопасными для многопоточного использования извне (обычно с использованием блокировки *внутри* ваших методов, как в приведенном выше примере). Пользователям вашей библиотеки намного сложнее, а может, и невозможно, ставить блокировку *вокруг* обращения к вашим статическим свойствам и методам, потому что они не знают, откуда еще ваши методы могут быть вызваны.

8.13. Оператор using

Оператор `using` получает один или несколько ресурсов, выполняет оператор, а затем освобождает эти ресурсы.

> **ЭРИК ЛИППЕРТ**
>
> В спецификации языка явно указано, что оператор `using` должен гарантировать получение ресурса и своевременное освобождение *ресурса*. Обычно это касается некоторого неуправляемого ресурса из операционной системы, такого как дескриптор файла. Правильным будет освободить такой ресурс как можно быстрее. Некоторые другие программы могут захотеть читать файл в одно время с вами. Я рекомендую не использовать оператор `using` для обеспечения устойчивости программы. Например, иногда встречается код наподобие этого:
>
> ```
> using(new TemporarilyStopReportingErrors()) AttemptSomething();
> ```
>
> В нем `TemporarilyStopReportingErrors` — это тип, конструктор которого в качестве побочного эффекта отключает сообщения об ошибках, а его деструктор включает их снова. Я считаю, что этот широко распространенный код является порочной практикой. Побочный эффект в программе — это не ресурс, и использование побочных эффектов в конструкторах и деструкторах является плохой идеей. Я бы написал этот код, используя оператор `try-finally`.

```
оператор-using
   using   (   получение-ресурса   )   вложенный-оператор

получение-ресурса:
   объявление-локальной-переменной
   выражение
```

Ресурс — это класс или структура, которая реализует интерфейс `System.IDisposable`, содержащий один метод `Dispose` без параметров. Код, использующий этот ресурс, может вызвать метод `Dispose` и сообщить, что ресурс больше не нужен. Если метод `Dispose` не был вызван, то через некоторое время произойдет автоматическое освобождение ресурса вследствие работы сборщика мусора.

> **ДЖОЗЕФ АЛБАХАРИ**
>
> Вызов метода `Dispose` в любом случае не влияет на сборку мусора. Объект становится доступным для сборщика мусора тогда и только тогда, когда на него нет ни одной ссылки. Также и сборщик мусора не влияет на освобождение ресурса, потому что он не вызывает `Dispose`, пока вы не напишете финализатор (деструктор), который будет вызывать его явным образом.
>
> Два действия, которые обычно выполняют внутри метода `Dispose`, — это освобождение неуправляемых ресурсов и вызов `Dispose` для объектов, которыми «обладает» данный или на которые есть ссылки из него. Освободить неуправляемые ресурсы также можно из финализатора, хотя это означает ожидание выполнения сборщика мусора в течение некоторого не известного промежутка времени. Вот поэтому интерфейс `IDispose` и существует.

Если формой *получения-ресурса* является *объявление-локальной-переменной*, то ее типом может быть или `dynamic` или тип, который можно неявно привести к `System.IDisposable`. Если формой *получения-ресурса* является *выражение*, оно должно неявно приводиться к `System.IDisposable`.

Объявленные при *получении-ресурса* локальные переменные доступны только для чтения и должны иметь инициализатор. Попытка изменить такую локальную переменную в коде вложенного оператора с помощью присваивания, операций `++` или `--` вызовет ошибку компиляции. Ошибка компиляции выдается при попытке передать эту переменную как `ref` или `out` параметр, а также при получении ее адреса.

Оператор `using` преобразуется в три этапа: получение ресурса, его использование и его освобождение. Использование ресурса неявно заключено в оператор `try` с блоком `finally`. В этом блоке `finally` происходит освобождение ресурса. Если в качестве ресурса будет `null`, то вызов `Dispose` не последует и исключение не будет выброшено. Если ресурс имеет тип `dynamic`, то во время получения ресурса будет выполнено неявное динамическое приведение (раздел 6.1.8) к `IDisposable` с целью проверки успешности этого преобразования перед использованием и освобождением.

Оператор `using` вида

```
using (ResourceType ресурс = выражение) оператор
```

соответствует одному из трех возможных расширений. Если `ResourceType` — необнуляемый тип-значение:

```
{
   ResourceType ресурс = выражение;
   try
   {
      оператор;
   }
   finally
   {
       ((IDisposable) ресурс).Dispose();
   }
}
```

Иначе, если `ResourceType` — обнуляемый тип-значение или ссылочный тип, не являющийся типом `dynamic`:

```
{
   ResourceType ресурс = выражение;
   try
   {
      оператор;
   }
   finally
   {
      if (resource != null) ((IDisposable)ресурс).Dispose();
   }
}
```

Иначе, если `ResourceType` – тип `dynamic`:

```
{
   ResourceType ресурс = выражение;
   IDisposable d = (IDisposable) ресурс;
   try
   {
      оператор;
   }
   finally
   {
      if (d != null) d.Dispose();
   }
}
```

Во всех этих случаях переменная `ресурс` доступна только для чтения внутри вложенного оператора, а переменная `d` не доступна и не видима внутри вложенного оператора.

> **ПИТЕР СЕСТОФТ**
>
> Благодаря этим правилам в операторе `using` можно объявлять доступные только для чтения локальные переменные таким образом:
>
> ```
> using (MyClass v = ...)
> using (MyStruct s = ...)
> {
> ...
> }
> ```
>
> На самом деле это единственный способ в языке C# объявить неизменную локальную переменную. Но это странный и очень некрасивый способ. Он работает только для классов и структур, которые поддерживают интерфейс `IDisposable`, и не применим для таких типов, как, например, `int`, `string` и т. д. С точки зрения описания языка удивительно, что в отличие от доступных только для чтения полей структуры (разделы 10.5.2 и 7.6.4), неизменяемая переменная ресурса `s` структурного типа `MyStruct` рассматривается как переменная, а не как значение. И таким образом вызов `s.SetX()` изменяет саму структуру `s`, а не ее копию. В целом этот пример показывает, что механизм описания неизменяемых переменных и параметров в C# существует, но в не пригодном для использования виде.

В разных реализациях языка допускается обрабатывать оператор `using` по-разному, например, для повышения быстродействия, но его поведение должно соответствовать упомянутым выше правилам.

Оператор `using` такого вида

`using` **`(`** *`выражение`* **`)`** *`оператор`*

имеет такие же три возможных расширения, но в этом случае `ResourceType` неявно является для `выражения` типом времени компиляции, а переменная `ресурс` недоступна и невидима внутри вложенного оператора.

> **ДЖОН СКИТ**
>
> Я обычно настаиваю на наличии фигурных скобок вокруг всего, и это распространяется и на оператор `using` тоже. Однако если вы получаете несколько ресурсов разных типов и вынуждены использовать несколько операторов `using`, то можете написать его только с одним набором скобок:

```
using (TextWriter output = File.CreateText("log.txt"))
using (TextReader input = File.OpenText("log.txt"))
{
   // копирование содержания из одного файла в другой
}
```

Так вы можете уменьшить степень вложенности операторов и сделать код более читабельным.

Когда *получение-ресурса* выполняется с помощью *описания-локальной-переменной*, можно выделить несколько ресурсов данного типа. Оператор `using` вида

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) оператор
```

эквивалентен последовательности вложенных операторов `using`:

```
using (ResourceType r1 = e1)
   using (ResourceType r2 = e2)
      ...
      using (ResourceType rN = eN)
         оператор
```

В следующем примере создается файл с названием `log.txt` и в него записываются две текстовые строки. Затем тот же самый файл считывается и его содержимое выводится на консоль.

```
using System;
using System.IO;
class Test
{
   static void Main()
   {
      using (TextWriter w = File.CreateText("log.txt"))
      {
         w.WriteLine("This is line one");
         w.WriteLine("This is line two");
      }
      using (TextReader r = File.OpenText("log.txt"))
      {
         string s;
         while ((s = r.ReadLine()) != null)
         {
            Console.WriteLine(s);
         }
      }
   }
}
```

Так как классы `TextWriter` и `TextReader` реализуют интерфейс `IDisposable`, в примере можно использовать оператор `using`, чтобы гарантировать корректное закрытие файла после операций записи и чтения.

КРИС СЕЛЛС

Вам следует почти всегда использовать оператор `using` при получении ресурса, который реализует `IDisposable`, если вы не сохраняете его между разными вызовами метода.

продолжение ⇨

Хотя сборщики мусора в .NET отлично справляются с освобождением памяти, все другие ресурсы должны контролироваться вами. Компилятор сгенерирует прекрасный код для освобождения ресурсов, но только если вы поместите выделение ресурсов в оператор `using`.

8.14. Оператор yield

БИЛЛ ВАГНЕР

Операторы `yield return` и `yield break` кажутся наиболее недооцененными в языке C#. Они чрезвычайно полезны при написании алгоритмов работы с последовательностями данных. Создание последовательностей, их фильтрация, объединение и другие алгоритмы могут быть прекрасно описаны с использованием этих операторов. Большинство конструкций LINQ to Objects используют оператор `yield`.

Если вы еще не знакомы с этими возможностями, потратьте время на их изучение, чтобы сделать их частью вашей ежедневной работы.

КРИС СЕЛЛС

Я согласен с Биллом. Я не мог оценить, насколько удобен оператор `yield return`, пока не увидел код наподобие этого:

```
IEnumerable<int> GetSomeNumbers()
{
   yield return 1;
   yield return 2;
   yield return 3;
}
```

Если вы поймете, что именно компилятор в этом случае делает за вас, то будете использовать эти возможности намного чаще.

Оператор `yield` используется в блоке итератора (раздел 8.2) для формирования значения перечислителя (раздел 10.14.4) или перечислимого объекта (раздел 10.14.5) итератора или для оповещения о конце цикла.

```
оператор-yield:
    yield   return   выражение   ;
    yield   break   ;
```

Стоит отметить, что `yield` — это не зарезервированное слово. Оно имеет специальное значение, только если находится непосредственно перед ключевыми словами `return` или `break`. В других контекстах слово `yield` может использоваться как идентификатор.

Есть несколько ограничений для использования оператора `yield`. Вот их список:

- При компиляции выдается ошибка, если оператор `yield` в любых его формах появится вне *тела-метода*, *тела-операции* или *тела-кода-доступа*.

- При компиляции выдается ошибка, если оператор `yield` в любых его формах появится внутри анонимной функции.
- При компиляции выдается ошибка, если оператор `yield` в любых его формах появится в блоке `finally` оператора `try`.
- При компиляции выдается ошибка, если оператор `yield return` появится в операторе `try`, который имеет блоки `catch`.

Следующий пример показывает допустимое и недопустимое использование оператора `yield`:

```
delegate IEnumerable<int> D();
IEnumerator<int> GetEnumerator()
{
   try
   {
      yield return 1; // Правильно
      yield break;    // Правильно
   }
   finally
   {
      yield return 2; // Ошибка: yield находится в finally
      yield break;    // Ошибка: yield находится в finally
   }
   try
   {
      yield return 3; // Ошибка: yield return находится в try...catch
      yield break;    // Правильно
   }
   catch
   {
      yield return 4; // Ошибка: yield return находится в try...catch
      yield break;    // Правильно
   }

   D d = delegate
   {
      yield return 5; // Ошибка: yield находится в теле анонимной функции
   };
}
int MyMethod()
{
   yield return 1;    // Ошибка: ошибочный тип возврата для блока итератора
}
```

Должно существовать неявное приведение (раздел 6.1) из типа выражения в операторе `yield return` к выходному типу итератора (раздел 10.14.3).

Оператор `yield return` выполняется следующим образом:

- Выражение в операторе вычисляется, неявно преобразуется к выходному типу и присваивается свойству `Current` перечислителя.
- Выполнение блока итератора приостанавливается. Если оператор `yield return` находится внутри одного или нескольких блоков `try`, то соответствующие им блоки `finally` *не* выполняются.

- Метод `MoveNext` перечислителя возвращает в точку вызова `true`, показывая, что перечислитель успешно переместился на следующий элемент.

Следующий вызов метода перечислителя `MoveNext` возобновит выполнение блока итератора с того места, где он был приостановлен последний раз.

Оператор `yield break` выполняется следующим образом:

- Если оператор `yield break` находится внутри одного или нескольких блоков `try` с соответствующими им блоками `finally`, управление передается блоку `finally` самого внутреннего оператора `try`. Если управление достигает конечной точки блока `finally`, управление передается блоку `finally` объемлющего оператора `try`. Этот процесс повторяется, пока не будут выполнены блоки `finally` всех операторов `try`.
- Управление возвращается в точку вызова блока итератора. Это либо метод `MoveNext`, либо метод `Dispose` перечислителя.

Так как оператор `yield break` безусловно передает управление, то его конечная точка недоступна.

КРИС СЕЛЛС

Я иногда забываю, что `yield return` — это не то же самое, что `return`, ведь код после `yield return` может выполняться. В этом примере код после первого `return` никогда не выполняется:

```
int F()
{
   return 1;
   return 2; // Никогда не будет выполняться
}
```

В отличие от этого примера, здесь код после первого `yield return` может быть выполнен:

```
IEnumerable<int> F() {
yield return 1;
yield return 2; // может быть выполнен
}
```

Это часто запутывает меня в операторе `if`:

```
IEnumerable<int> F()
{
   if(...) { yield return 1; } // я полагаю, что должно быть возвращено
                               // только это значение
   yield return 2;             // Ой!
}
```

В этом случае помогает помнить, что на операторе `yield return` выполнение метода не заканчивается (в отличие от `return`).

КРИСТИАН НЕЙГЕЛ

Начиная с C# 1.0 стало удобно использовать перечислители (оператор `foreach`). Начиная с C# 2.0 стало удобно создавать перечислители (оператор `yield`).

Глава 9

Пространства имен

Программы на языке C# используют пространства имен. Пространство имен служит как для «внутренней» организации программы, так и для «внешней» — в качестве способа предоставления элементов программы другим программам.

Директивы `using` (раздел 9.4) облегчают использование пространств имен.

> **БИЛЛ ВАГНЕР**
>
> Помните, что пространства имен используются для логической организации кода. В одной сборке может быть несколько пространств имен, и одно и то же пространство имен можно объявить в нескольких сборках.

9.1. Единица компиляции

Единица-компиляции определяет общую структуру исходного файла. Он может содержать ноль или более *директив* `using`, за которыми следует ноль или более *глобальных-атрибутов*, а далее — ноль или более *объявлений-элементов-пространства-имен*.

единица-компиляции:
    директивы-внешнего-псевдонима$_{opt}$ *директивы-using*$_{opt}$ *глобальные-атрибуты*$_{opt}$
        объявления-элементов-пространства-имен

Программа на языке C# состоит из одной или нескольких единиц компиляции, каждая из которых находится в отдельном исходном файле. Когда программа на языке C# компилируется, то все ее единицы компиляции обрабатываются совместно. Таким образом, единицы компиляции могут зависеть друг от друга, возможно, циклически.

> **КРИС СЕЛЛС**
>
> Одна эта возможность позволяет сделать работу на языке C# раз в десять проще, чем на C или C++.

Директивы-using единицы компиляции влияют на *глобальные-атрибуты* и *объявления-элементов-пространства-имен* только внутри этой единицы компиляции и не действуют на другие единицы компиляции.

> **ДЖОН СКИТ**
>
> Чаще всего я считаю эту возможность полезной. Но иногда, когда я делаю программу «на скорую руку», только чтобы проверить несколько строчек кода, я бы предпочел не импортировать целую пачку пространств имен. Было бы интересно представить, какой эффект имело бы наличие конструкций `using` на уровне проекта (аналогично ссылкам на сборки) в альтернативной реальности.

> **ЭРИК ЛИППЕРТ**
>
> Точка зрения Джона вполне обоснована. Стандартные действия, необходимые для написания простейшей программы на языке C#, неоправданно сложны. Хотя язык C# не является языком сценариев, удобно, когда программа из одной строки действительно представляет собой всего одну строку кода, как в Jscript.

Глобальные-атрибуты (раздел 17) единицы компиляции позволяют определить атрибуты для сборки и модуля. Сборки и модули являются физическими контейнерами для типов. Сборка может состоять из нескольких отдельных модулей.

Объявление-элементов-пространства-имен каждой единицы компиляции добавляет элементы в единое пространство имен, называемое глобальным. Например файл `A.cs`:

```
class A {}
```

файл `B.cs`:

```
class B {}
```

Эти две единицы компиляции относятся к единому глобальному пространству имен, в этом случае путем объявления двух классов с полными именами `A` и `B`. Так как обе единицы компиляции относятся к одной и той же области объявлений, то при наличии в них элементов с одинаковыми именами при компиляции произойдет ошибка.

9.2. Объявления пространств имен

Объявление-пространства-имен состоит из ключевого слова `namespace`, за которым следуют имя и тело пространства имен. В конце может присутствовать точка с запятой, но ее наличие не обязательно.

```
объявление-пространства-имен:
   namespace   уточненное-имя   тело-пространства-имен opt   ; opt

уточненное-имя:
   идентификатор
   уточненное-имя   .   идентификатор

тело-пространства-имен:
   { директивы-внешнего-псевдонима opt   директивы-using opt
            объявления-элементов-пространства-имен opt   }
```

Объявление-пространства-имен может встречаться на самом верхнем уровне в *единице-компиляции* или являться элементом другого *объявления-пространства-имен*. Когда *объявление-пространства-имен* находится на самом верхнем уровне в *единице-компиляции*, то оно становится элементом глобального пространства имен. Когда *объявление-пространства-имен* находится внутри другого *объявления-пространства-имен*, то внутреннее пространство имен становится элементом внешнего. В любом случае, имя пространства имен должно быть уникальным в пределах того пространства, в котором оно объявлено.

Пространство имен неявно считается `public`, и его объявление не может содержать никаких модификаторов доступа.

В *теле-пространства-имен* необязательные директивы `using` импортируют имена других пространств имен и типов, позволяя ссылаться на них напрямую, без указания полного имени. Необязательные *объявления-элементов-пространства-имен* добавляют элементы в область объявлений данного пространства имен. Заметьте, что все директивы `using` должны предшествовать любым объявлениям элементов.

Уточненное-имя в *объявлении-пространства-имен* может быть как простым именем, так и последовательностью имен, разделенных точками. Такой способ объявления позволяет определить вложенное пространство имен без использования нескольких вложенных объявлений. Например, такое объявление:

```
namespace N1.N2
{
   class A { }
   class B { }
}
```

эквивалентно

```
namespace N1
{
   namespace N2
   {
      class A { }
      class B { }
   }
}
```

Пространства имен являются расширяемыми. Два объявления пространства имен с одним и тем же полностью уточненным именем относятся к одной области объявлений (раздел 3.3).

Например:

```
namespace N1.N2
{
   class A { }
}
namespace N1.N2
{
   class B { }
}
```

Эти два объявления пространства имен относятся к одной области объявлений, и они содержат два класса с полными именами `N1.N2.A` и `N1.N2.B`. Так как

оба этих класса находятся в одном пространстве имен, то если бы они назывались одинаково, при компиляции произошла бы ошибка.

9.3. Внешние псевдонимы

Директива-внешнего-псевдонима объявляет идентификатор, который служит псевдонимом (синонимом) для пространства имен. Пространство имен, на которое указывает псевдоним, является внешним по отношению к исходному коду программы. Псевдоним может использоваться также для пространств имен, вложенных в пространство, имеющее псевдоним.

```
директивы-внешнего-псевдонима:
   директива-внешнего-псевдонима
   директивы-внешнего-псевдонима  директива-внешнего-псевдонима

директива-внешнего-псевдонима:
   extern alias  идентификатор  ;
```

Область действия *директивы-внешнего-псевдонима* распространяется на *директивы* `using`, *глобальные-атрибуты* и *объявления-элементов-пространства-имен* в той единице компиляции или теле пространства имен, где он непосредственно объявлен.

В единице компиляции или теле пространства имен, содержащих *директиву-внешнего-псевдонима*, идентификатор псевдонима используется для ссылки на пространство имен, которому он сопоставлен. Если в качестве идентификатора используется слово `global`, при компиляции произойдет ошибка.

Директива-внешнего-псевдонима делает псевдоним доступным внутри единицы компиляции или тела пространства имен, но он не создает новых элементов для соответствующей области объявлений. Иными словами, *директива-внешнего-псевдонима* не транзитивна, она действует только в пределах единицы компиляции или тела пространства имен, в котором она встречается.

В следующей программе объявляется и используется два внешних псевдонима `X` и `Y`, каждый из них представляет собой верхний уровень разных иерархий пространств имен.

```
extern alias X;
extern alias Y;
class Test
{
   X::N.A a;
   X::N.B b1;
   Y::N.B b2;
   Y::N.C c;
}
```

В программе объявляются внешние псевдонимы `X` и `Y`, но их фактическое определение находится за пределами кода. Теперь на элементы с одним и тем же именем класса `N.B` можно ссылаться так: `X.N.B` и `Y.N.B` или через уточняющий псевдоним `X::N.B` и `Y::N.B`. Если в программе будет объявлен внешний псевдоним, который нигде не определен, при компиляции произойдет ошибка.

9.4. Директивы using

Директивы `using` облегчают использование пространств имен и определенных в них типов. Директивы `using` влияют на процесс разрешения *имен-пространств-имен-или-типов* (раздел 3.8) и *простых-имен* (раздел 7.6.2), но в отличие от объявлений директивы `using` не создают новых элементов в соответствующей области объявлений единиц компиляции или пространств имен, в которых используются эти директивы.

```
директивы-using:
   директива-using
   директивы-using   директива-using

директива-using:
   using-директива-псевдонима
   using-директива-пространства-имен
```

Using-директива-псевдонима (раздел 9.4.1) вводит псевдоним для пространства имен или типа.

Using-директива-пространства-имен (раздел 9.4.2) импортирует типы пространства имен.

Область действия директивы `using` распространяется на объявление элементов пространства имен в данной единице компиляции или теле пространства имен. Область действия директивы `using` не распространяется на соседние директивы `using`. Таким образом, директивы `using` не влияют друг на друга, и их порядок не имеет значения.

9.4.1. Using-директива псевдонима

Using-директива-псевдонима вводит идентификатор, который будет служить псевдонимом пространства имен или типа.

```
using-директива-псевдонима:
   using   идентификатор   =   пространство-имен-или-тип   ;
```

Using-директива-псевдонима позволяет использовать псевдоним как ссылку на соответствующее пространство имен или тип в единице компиляции или теле пространства имен, где он объявлен. Например:

```
namespace N1.N2
{
   class A { }
}
namespace N3
{
   using A = N1.N2.A;
   class B: A { }
}
```

В этом примере при объявлении элементов пространства имен `N3` псевдоним `A` используется для класса `N1.N2.A`, и, следовательно, класс `B` является наследником класса `N1.N2.A`. То же самое можно получить, если использовать псевдоним `R` для пространства имен `N1.N2` и ссылаться на `R.A`:

```
namespace N3
{
   using R = N1.N2;
   class B: R.A { }
}
```

Идентификатор в `using` директиве псевдонима должен быть уникальным в пределах единицы компиляции или пространства имен, в которых он непосредственно объявлен. Например:

```
namespace N3
{
   class A { }
}
namespace N3
{
   using A = N1.N2.A; // Ошибка: A уже используется
}
```

В этом примере пространство имен `N3` уже содержит элемент с именем `A`, следовательно, при компиляции произойдет ошибка. Ошибка будет также и при наличии двух или более директив `using` для псевдонимов с одинаковыми именами в пределах одной единицы компиляции или тела пространства имен.

Using-директива-псевдонима делает его доступным в определенной единице компиляции или теле пространства имен, но она не вводит новых элементов в соответствующую область объявлений. Иными словами, *using-директива-псевдонима* действует только внутри единицы компиляции или тела пространства имен, в котором она находиться.

В примере

```
namespace N3
{
   using R = N1.N2;
}
namespace N3
{
   class B: R.A { } // Ошибка: R не определено
}
```

Область действия `R` распространяется только на объявления элементов в теле пространства имен, где объявлена директива, таким образом, `R` не известно внутри второго объявления пространства имен. Однако, если *using-директиву-псевдонима* `R` поместить непосредственно в единицу компиляции, она будет доступна в обоих объявлениях пространства имен:

```
using R = N1.N2;
namespace N3
{
   class B: R.A {}
}
namespace N3
{
   class C: R.A {}
}
```

Как и обычный элемент, имя, объявленное с помощью *using-директивы-псевдонима*, будет скрыто другим элементом с таким же именем, объявленным во вложенных областях определения. Например:

```
using R = N1.N2;
namespace N3
{
   class R {}
   class B: R.A {} // Ошибка: R не содержит элемента с именем A
}
```

Ссылка на `R.A` при объявлении класса `B` вызовет ошибку компиляции, потому что `R` ссылается на `N3.R`, а не на `N1.N2`.

Порядок следования *using-директив-псевдонима* не имеет значения, и разрешение *имени-пространства-имен-или-типа*, на которые ссылается псевдоним, не зависят от этой или других директив `using` в данной единице компиляции или теле пространства имен. Иными словами, *имя-пространства-имен-или-типа using-директивы-псевдонима* разрешается так, как будто в содержащей его единице компиляции или теле пространства имен нет директив `using`. Однако на *using-директиву-псевдонима* могут оказывать влияние *директивы-внешнего-псевдонима*, которые содержатся в данной единице компиляции или теле пространства имен. Например:

```
namespace N1.N2 { }
namespace N3
{
   extern alias E;
   using R1 = E.N;    // правильно
   using R2 = N1;     // правильно
   using R3 = N1.N2;  // правильно
   using R4 = R2.N2;  // Ошибка: R2 не определено
}
```

Последняя директива `using` при компиляции вызовет ошибку, потому что на нее не влияет первая из *using-директив-псевдонима*. Первая *using-директива-псевдонима* не вызовет ошибки, так как находится в области действия внешнего псевдонима `E`.

Using-директива-псевдонима может создать псевдоним любого пространства имен или типа, включая то пространство имен, в котором она находится, или вложенных в него пространств имен или типов.

Использование псевдонима в коде абсолютно эквивалентно использованию полного имени пространства имен или типа, на который он ссылается. Например:

```
namespace N1.N2
{
   class A { }
}
namespace N3
{
   using R1 = N1;
   using R2 = N1.N2;
   class B
```

продолжение ☞

```
   {
      N1.N2.A a;   // ссылается на N1.N2.A
      R1.N2.A b;   // ссылается на N1.N2.A
      R2.A c;      // ссылается на N1.N2.A
   }
}
```

обращения `N1.N2.A`, `R1.N2.A`, `R2.A` абсолютно эквивалентны, и все они ссылаются на класс с полным именем `N1.N2.A`.

Псевдоним может ссылаться на закрытый сконструированный тип, но не может ссылаться на неограниченный обобщенный тип без указания аргументов типа. Например:

```
namespace N1
{
   class A<T>
   {
      class B {}
   }
}
namespace N2
{
    using W = N1.A;   // Ошибка: нельзя ссылаться на неограниченный обобщенный тип
    using X = N1.A.B; // Ошибка: нельзя ссылаться на неограниченный обобщенный тип
    using Y = N1.A<int>;  // Правильно: можно ссылаться на закрытый
                          // сконструированный тип
    using Z<T> = N1.A<T>; // Ошибка: using-директива псевдонима
                          // не может иметь параметров-типов
}
```

9.4.2. Using-директива пространства имен

Using-директива-пространства-имен импортирует типы, которые в нем содержатся, в данную единицу компиляции или тело пространства имен. Это позволяет ссылаться на типы из импортируемого пространства по их идентификаторам, без необходимости уточнения имени.

```
using-директива-пространства-имен:
   using   имя-пространства-имен   ;
```

При описании элементов единицы компиляции или тела пространства имен, содержащего *using-директиву-пространства-имен*, можно использовать имена содержащихся в этом пространстве имен типов непосредственно. Например:

```
namespace N1.N2
{
   class A { }
}
namespace N3
{
   using N1.N2;
   class B: A { }
}
```

В этом примере внутри объявления элементов пространства имен `N3` можно напрямую ссылаться на тип `N1.N2`, то есть класс `N3.B` наследуется от `N1.N2.A`.

Using-директива-пространства-имен импортирует типы из этого пространства имен, но не импортирует типы из вложенных в него пространств имен. Например:

```
namespace N1.N2
{
   class A { }
}

namespace N3
{
   using N1;
   class B: N2.A { } // Ошибка: N2 не определено
}
```

Using-директива-пространства-имен `N1` импортирует все типы из него, но не из вложенного в него пространства имен. Следовательно, использование ссылки `N2.A` в объявлении класса `B` при компиляции вызовет ошибку, так как нет ни одного элемента с именем `N2`.

В отличие от *using-директив-псевдонимов*, *using-директива-пространства-имен* может импортировать типы, имена которых совпадают с именами типов, определенными внутри данной единицы компиляции или тела пространства имен. В таком случае эти типы из импортируемого пространства имен будут скрыты. Например:

```
namespace N1.N2
{
   class A { }
   class B { }
}

namespace N3
{
   using N1.N2;
   class A { }
}
```

Здесь при объявлении элементов пространства имен `N3` ссылка на тип `A` будет указывать на `N3.A`, а не на тип `N1.N2.A`.

Когда имена типов из нескольких импортируемых пространств имен совпадают, то ссылка на такой тип считается неоднозначной. Например:

```
namespace N1
{
   class A { }
}
namespace N2
{
   class A { }
}
namespace N3
{
   using N1;
   using N2;
   class B : A { } // Ошибка: A неоднозначно
}
```

оба пространства имен `N1` и `N2` содержат элемент `A`, и они оба импортируются в `N3`, поэтому ссылка на тип `A` вызовет ошибку компиляции. В такой ситуации конфликт можно разрешить либо использованием полного имени при ссылке на тип `A`, либо с помощью *using-директивы-псевдонима*, указывающей на конкретный тип `A`. Пример:

```
namespace N3
{
   using N1;
   using N2;
   using A = N1.A;
   class B: A { } // A означает N1.A
}
```

Как и *using-директива-псевдонима*, *using-директива-пространства-имен* не добавляет новых элементов в область объявлений данной единицы компиляции или тела пространства имен и действует только внутри них.

Имя-пространства-имен, задаваемое в *using-директиве-пространства-имен,* разрешается так же, как *имя-пространства-имен-или-типа using-директивы-псевдонима*. То есть *using-директивы-пространства-имен* в единице компиляции или в теле пространства имен не влияют друг на друга и могут быть записаны в любом порядке.

9.5. Элементы пространства имен

Объявление-элемента-пространства-имен может быть либо *объявлением-пространства-имен* (раздел 9.2), либо *объявлением-типа* (раздел 9.6).

```
объявление-элементов-пространства-имен:
   объявление-элемента-пространства-имен
   объявление-элементов-пространства-имен   объявление-элемента-пространства-имен

объявление-элемента-пространства-имен:
   объявление-пространства-имен
   объявление-типа
```

Единица компиляции или тело пространства имен могут содержать *объявления-элементов-пространства-имен,* которые добавляют новые элементы в соответствующую область объявлений.

9.6. Объявление типов

Объявление-типа бывает *объявлением-класса* (раздел 10.1), *объявлением-структуры* (раздел 11.1), *объявлением-интерфейса* (раздел 13.1), *объявлением-перечисления* (раздел 14.1) или *объявлением-делегата* (раздел 15.1).

```
объявление-типа:
   объявление-класса
   объявление-структуры
   объявление-интерфейса
   объявление-перечисления
   объявление-делегата
```

Объявление-типа может находиться на самом верхнем уровне единицы компиляции или являться элементом пространства имен, класса или структуры.

Если объявление типа `T` находится на самом верхнем уровне единицы компиляции, полным именем этого типа является `T`. Если же объявление типа `T` находится внутри пространства имен, класса или структуры, его полным именем является `N.T`, где `N` – полное имя того пространства имен, класса или структуры, в котором тип `T` объявлен.

Тип, объявленный внутри класса или структуры, называют вложенным типом (раздел 10.3.8).

Допустимые модификаторы доступа, а также модификаторы, принятые по умолчанию для данного типа, зависят от контекста, в котором сделано это объявление (раздел 3.5.1):

- Типы, объявленные в единице компиляции или пространствах имен, могут иметь модификаторы `public` или `internal`. По умолчанию такой тип считается `internal`.
- Типы, объявленные в классе, могут иметь модификаторы `public`, `protected internal`, `internal` или `private`. По умолчанию они являются `private`.
- Типы, объявленные в структурах, могут иметь модификаторы `public`, `internal` или `private`. По умолчанию они являются `private`.

9.7. Спецификатор псевдонима пространства имен

Спецификатор псевдонима пространства имен `::` позволяет гарантировать, что поиск имен для типа не будет зависеть от добавления новых типов и элементов. Спецификатор псевдонима пространства имен всегда используется между двумя идентификаторами, называемыми левосторонним и правосторонним идентификаторами.

В отличие от обычного спецификатора `.`, левосторонний идентификатор в спецификаторе `::` считается только либо внешним псевдонимом, либо using-псевдонимом.

Уточненный-псевдоним определяется следующим образом:

```
уточненный-псевдоним:
   идентификатор   ::   идентификатор   список-аргументов-типов_opt
```

Уточненный-псевдоним может использоваться как *имя-пространства-имен-или-типа* (раздел 3.8) или как левый операнд при *доступе-к-элементу* (раздел 7.6.4).

Уточненный-псевдоним имеет одну из двух форм:

- `N :: I <A1,…,Ak>` , где `N` и `I` – это идентификаторы, а `<A1,…,Ak>` – список аргументов-типов (`k` всегда больше или равно 1).
- `N :: I` , где `N` и `I` – это идентификаторы (в этом случае `k` всегда 0).

В соответствии с этими формами значение *уточненного-псевдонима* определяется следующим образом:

- Если `N` – это идентификатор `global`, тогда для `I` ищется глобальное пространство имен следующим образом:
  - Если глобальное пространство имен содержит пространство имен `I` и `k` равно нулю, *уточненный-псевдоним* ссылается на это пространство имен.
  - Иначе, если глобальное пространство имен содержит необобщенный тип с именем `I` и `k` равно нулю, *уточненный-псевдоним* ссылается на этот тип.
  - Иначе, если глобальное пространство имен содержит тип `I`, имеющий `k` параметров-типов, *уточненный-псевдоним* ссылается на тип, сконструированный с данными аргументами-типами.
  - Иначе *уточненный-псевдоним* считается неопределенным и выдается ошибка компиляции.
- Иначе, начиная с объявления пространства имен (раздел 9.2), в котором был объявлен *уточненный-псевдоним* (при его наличии), и до самого верхнего объявления пространства имен (при их наличии), и заканчивая единицей компиляции, в которой объявлен псевдоним, выполняются следующие шаги, пока сущность не будет определена:
  - Если в пространстве имен или единице компиляции содержится *using-директива-псевдонима*, который связывает `N` с каким-либо типом, то *уточненный-псевдоним* считается неопределенным, и возникает ошибка компиляции.
  - Иначе, если пространство имен или единица компиляции содержат *директиву-внешнего-псевдонима* или *using-директиву-псевдонима*, связывающего `N` с каким-либо пространством имен, то:
    - Если пространство имен, связанное с `N`, содержит пространство имен `I` и `K` равно нулю, *уточненный-псевдоним* ссылается на это пространство имен.
    - Иначе, если пространство имен, связанное с `N`, содержит необобщенный тип `I` и `K` равно нулю, *уточненный-псевдоним* ссылается на этот тип.
    - Иначе, если пространство имен, связанное с `N`, содержит тип `I`, который имеет `K` параметров-типов, то *уточненный-псевдоним* ссылается на тип, сконструированный с данными параметрами-типами.
    - Иначе *уточненный-псевдоним* не определен, и при компиляции произойдет ошибка.
- Иначе *уточненный-псевдоним* не определен, и при компиляции произойдет ошибка.

Стоит отметить, что использование спецификатора псевдонима пространства имен с псевдонимом, ссылающимся на тип, вызывает ошибку компиляции. Еще заметим, что если идентификатор `N` является `global`, то поиск производится в глобальном пространстве имен, даже если определен псевдоним, связывающий `global` с каким-либо пространством имен или типом.

9.7.1. Уникальность псевдонимов

Каждая единица компиляции и тело пространства имен имеют отдельные области объявлений для внешних псевдонимов и `using`-псевдонимов. Таким образом, хотя имена внешних псевдонимов и `using`-псевдонимов должны быть уникальными для данного набора внешних псевдонимов и `using`-псевдонимов, объявленных непосредственно в единице компиляции или теле пространства имен, разрешается, чтобы имя псевдонима совпадало с именем типа или пространства имен, поскольку к нему обращаются только через спецификатор `::`.

В примере

```
namespace N
{
   public class A { }
   public class B { }
}
namespace N
{
   using A = System.IO;
   class X
   {
      A.Stream s1;    // Ошибка: A неоднозначно
      A::Stream s2;   // Правильно
   }
}
```

у имени `A` есть два возможных значения в теле второго пространства имен, потому что и класс `A`, и `using`-псевдоним `A` находятся в области видимости. По этой причине использование `A` в уточненном имени `A.stream` неоднозначно и вызывает ошибку компиляции. Однако использование `A` со спецификатором `::` ошибки не вызывает, поскольку `A` рассматривается только как псевдоним пространства имен.

Глава 10
Классы

Класс — это структура данных, которая может содержать элементы данных (константы и поля), функциональные элементы (методы, свойства, события, индексаторы, операции, конструкторы экземпляров, деструкторы и статические конструкторы), а также вложенные типы. Классы поддерживают наследование — механизм, с помощью которого производный класс может расширять и уточнять базовый.

10.1. Объявления классов

Объявление-класса — это *объявление-типа* (раздел 9.6), объявляющее новый класс.

```
объявление-класса:
   атрибуты_opt   модификаторы-класса_opt   partial_opt   class   идентификатор
        список-параметров-типов_opt   базовый-класс_opt
            ограничения-на-параметры-типы_op   тело-класса   ;_opt
```

Объявление-класса состоит из необязательного набора *атрибутов* (раздел 17), необязательного набора *модификаторов-класса* (раздел 10.1.1), необязательного модификатора `partial`, за которыми следуют ключевое слово `class` и *идентификатор*, определяющий имя класса, за которыми идут необязательный *список-параметров-типов* (раздел 10.1.3), необязательная спецификация *базового-класса* (раздел 10.1.4), необязательный набор *ограничений-на-параметры-типы* (раздел 10.1.5), *тело-класса* (раздел 10.1.6) и необязательная точка с запятой.

При наличии в объявлении класса *ограничений-на-параметры-типы* оно также должно содержать и *список-параметров-типов*.

Объявление класса, в котором присутствует *список-параметров-типов*, является **объявлением обобщенного класса**. Кроме того, любой класс, вложенный в объявление обобщенного класса или обобщенной структуры, также представляет собой объявление обобщенного класса, поскольку для создания сконструированного типа внешнему типу должны быть переданы параметры-типы.

10.1.1 Модификаторы класса

Объявление-класса может содержать необязательную последовательность модификаторов класса:

```
модификаторы-класса:
   модификатор-класса
   модификаторы-класса   модификатор-класса
```

```
модификатор-класса:
   new
   public
   protected
   internal
   private
   abstract
   sealed
   static
```

Если один и тот же модификатор встречается в объявлении класса несколько раз, возникает ошибка компиляции.

Для вложенных классов допускается использование модификатора `new`. Как описано в разделе 10.3.4, он указывает на то, что класс скрывает унаследованный элемент с таким же именем. Если модификатор `new` используется в объявлении класса, которое не является вложенным, возникает ошибка компиляции.

Модификаторы `public`, `protected`, `internal` и `private` управляют доступом к классу. В зависимости от контекста, в котором находится объявление класса, некоторые из этих модификаторов могут быть недопустимы (раздел 3.5.1).

Модификаторы `abstract`, `sealed` и `static` рассматриваются в следующих разделах.

10.1.1.1. Абстрактные классы

Модификатор `abstract` используется для обозначения того, что класс является незаконченным и предназначен для использования только в качестве базового класса. Абстрактный класс отличается от неабстрактного в следующем:

- Невозможно явно создать экземпляр абстрактного класса, и использование операции `new` с абстрактным классом приводит к ошибке компиляции. Несмотря на то что возможно существование переменных и значений, имеющих абстрактные типы времени компиляции, они обязательно будут иметь значения `null` или содержать ссылки на экземпляры неабстрактных классов, унаследованных от абстрактных типов.
- В абстрактном классе допускается (но не требуется) наличие абстрактных элементов.
- Абстрактный класс не может быть бесплодным (sealed).

Когда от абстрактного класса наследуется неабстрактный, последний должен содержать фактические реализации всех унаследованных абстрактных элементов, тем самым переопределяя их. Пример:

```
abstract class A
{
    public abstract void F();
}

abstract class B: A
{
    public void G() {}
}
```

продолжение ➪

```
class C: B
{
   public override void F() {
      // Фактическая реализация F
   }
}
```

В этом примере абстрактный класс `A` вводит абстрактный метод `F`. Класс `B` вводит дополнительный метод `G`, но поскольку он не предоставляет реализацию `F`, `B` также должен быть объявлен как абстрактный. Класс `C` переопределяет `F` и предоставляет его фактическую реализацию. Так как в `C` нет абстрактных методов, он может (но не обязан) быть неабстрактным.

10.1.1.2. Бесплодные классы

Модификатор `sealed` используется, чтобы сделать невозможным наследование класса. Если класс с этим модификатором указан в качестве базового для другого класса, возникает ошибка компиляции.

Бесплодный класс не может одновременно быть абстрактным.

Модификатор `sealed` главным образом используется для предотвращения непреднамеренного наследования, но он также задействует некоторую оптимизацию времени выполнения. В частности, для экземпляра бесплодного класса становится возможным преобразовать вызовы виртуальных функциональных элементов в невиртуальные, поскольку заведомо известно, что этот класс не будет иметь наследников.

ДЖОН СКИТ

Решение не делать классы бесплодными (а методы виртуальными) по умолчанию всегда было предметом горячих споров. Я согласен с аксиомой «Предусмотри наследование в проекте или запрети его», но для обоих случаев есть свои аргументы. Предоставление такого существенного умолчания является довольно странным: даже несмотря на то, что обычно я *делаю* классы бесплодными, если такая мысль приходит мне в голову, слишком легко попросту об этом забыть. Данный факт, очевидно, не влияет на смысл, но я уверен, что он оказывает влияние на уже использующийся код.

ДЖЕСС ЛИБЕРТИ

Я рискну не согласиться с Джоном.

Очевидно, что вы не можете точно предвидеть, какими будут требования даже через несколько месяцев. Поэтому практические соображения диктуют нам оставлять архитектуру проекта более опасной, не реализуя или почти не реализуя возможности до тех пор, пока они фактически не понадобятся, и избегая закрывать пути, которые, как нам кажется, никогда не потребуются.

Модификатор `sealed` — это минное поле, использующееся, чтобы заявить: «Вам никогда не понадобится идти туда». Я не думаю, что вы можете это знать или должны делать какие-либо предположения.

10.1.1.3. Статические классы

Модификатор `static` используется, чтобы пометить объявляемый класс как **статический класс**. Невозможно создать экземпляр статического класса, он не может использоваться в качестве типа и может содержать только статические элементы. Объявления методов расширения (раздел 10.6.9) могут находиться только в статическом классе.

На объявление статического класса накладываются следующие ограничения:

- Статический класс не может содержать модификаторы `sealed` или `abstract`. Однако заметьте, что поскольку от такого класса нельзя наследовать и нельзя создать его экземпляр, он ведет себя так же, как если бы одновременно был бесплодным и абстрактным.
- Статический класс не может содержать спецификацию *базового-класса* (раздел 10.1.4) и не может явно указывать базовый класс или список реализуемых интерфейсов. Статический класс неявно наследуется от типа `object`.
- Статический класс может содержать только статические элементы (раздел 10.3.7). Заметьте, что к статическим элементам также относятся константы и вложенные типы.
- Статический класс не может содержать элементы с видом доступа `protected` или `protected internal`.

Нарушение любого из этих ограничений приведет к ошибке компиляции.

У статического класса не существует конструкторов экземпляра. В нем невозможно объявить конструктор экземпляра и для него не создается конструктор экземпляра по умолчанию (раздел 10.11.4).

Элементы статического класса не становятся статическими автоматически, и их объявления должны явно содержать модификатор `static` (за исключением констант и вложенных типов). Когда класс вложен во внешний статический класс, вложенный класс не является статическим, если это не указано явно с помощью модификатора `static`.

> **МАРЕК САФАР**
>
> Для имитации статических классов в C# 1.0 приходилось использовать бесплодные классы с закрытыми конструкторами. Теперь это не требуется, так как статические классы предлагают более элегантный способ выразить данное намерение, к тому же компилятор обеспечивает проведение множества проверок на предмет недопустимых в статическом контексте операций.

10.1.1.3.1. Обращение к типам статических классов

Имя-пространства-имен-или-типа (раздел 3.8) может ссылаться на статический класс, если:

- оно представляет собой `T` в *имени-пространства-имен-или-типа* в форме `T.I`, или

- оно представляет собой `T` в *выражении-typeof* (раздел 7.5.11) в форме `typeof(T)`.

Первичное-выражение (раздел 7.5) может ссылаться на статический класс, если

- оно представляет собой `E` в *доступе-к-элементу-объекта* (раздел 7.5.4) в форме `E.I`.

В любых других контекстах обращение к статическому классу приводит к ошибке компиляции. Например, недопустимо использовать статический класс в качестве базового класса, составного типа (раздел 10.3.8) элемента, аргумента обобщенного типа или ограничения на параметр-тип. Кроме того, статический класс не может использоваться в типе массива, типе указателя, выражении `new`, выражении приведения типа, выражениях `is`, `as`, `sizeof`, а также в выражении значения по умолчанию.

10.1.2. Модификатор partial

Модификатор `partial` используется для обозначения того, что *объявление-класса* является объявлением частичного типа. Несколько объявлений частичного типа с одинаковым именем в охватывающем пространстве имен или объявлении типа объединяются в одно объявление типа в соответствии с правилами, приведенными в разделе 10.2.

Разделение объявления класса между несколькими сегментами текста программы может быть полезным, если эти сегменты создаются или поддерживаются в различных контекстах. Например, одна часть объявления класса может генерироваться автоматически, а другая создаваться вручную. Текстовое разделение этих частей предотвращает конфликты при обновлении каждой из них.

> **КРИС СЕЛЛЗ**
>
> Мне нравится возможность разделять частичные классы на часть, сгенерированную автоматически, и часть, созданную человеком. К сожалению, вы можете злоупотребить этой возможностью, разделив класс между более чем двумя файлами. Как человек, читающий код, я с трудом представляю, как можно следовать этой практике, и считаю ее крайне нежелательной.

10.1.3. Параметры-типы

Параметр-тип — это простой идентификатор, представляющий собой указатель места вставки аргумента-типа, передаваемого для создания сконструированного типа. Параметр-тип — это формальный указатель места вставки типа, который будет передан позже. Напротив, аргумент-тип (раздел 4.4.1) — это фактический тип, подставляемый на место параметра-типа при создании сконструированного типа.

```
список-параметров-типа:
  <   параметры-типа   >
```

```
параметры-типы:
   атрибуты_opt   параметр-тип
   параметры-типы  ,  атрибуты_opt   параметр-тип

параметр-тип:
   идентификатор
```

Каждый параметр-тип в объявлении класса определяет имя в области объявлений (раздел 3.3) этого класса. Таким образом, имя параметра-типа не может совпадать с именем другого параметра-типа или элемента, объявленного в этом классе. Оно также не может совпадать с именем самого типа.

10.1.4. Спецификация базового класса

Объявление класса может содержать спецификацию *базового-класса*, определяющую непосредственный базовый класс этого класса, а также интерфейсы (раздел 13), непосредственно им реализуемые.

```
базовый-класс:
   :  класс
   :  список-интерфейсов
   :  класс  ,  список-интерфейсов

список-интерфейсов:
   интерфейс
   список-интерфейсов  ,  интерфейс
```

Базовый класс, указанный в объявлении класса, может являться сконструированным типом класса (раздел 4.4). Сам базовый класс не может быть параметром-типом, однако он может задействовать параметры-типы, находящиеся в области видимости.

```
class Extend<V>: V {}        // Ошибка: параметр-тип используется
                             // в качестве базового класса
```

БИЛЛ ВАГНЕР

Я хотел бы, чтобы данное ограничение было убрано. Таким образом, появился бы замечательный способ создавать примеси (mixin). Я понимаю, что проблема очень сложна, поскольку `V` может содержать любые произвольные методы и свойства. В зависимости от конкретного типа, используемого в качестве `V` в закрытом обобщенном типе, `Extend<V>` может не скомпилироваться.

10.1.4.1. Базовые классы

Когда *класс* включен в список *базового-класса*, он определяет непосредственный базовый класс объявляемого класса. Если в объявлении класса отсутствует список *базового-класса* или если в этом списке перечислены только интерфейсы, предполагается, что непосредственным базовым классом является `object`. Класс наследует элементы от своего непосредственного базового класса, как описано в разделе 10.3.3.

Пример:

```
class A {}
class B: A {}
```

Здесь класс `A` является непосредственным базовым классом `B`, а `B` унаследован от `A`. Так как непосредственный базовый класс для `A` не указан явно, таким классом неявно является `object`.

Для сконструированного класса, если в объявлении обобщенного класса указан базовый класс, то для сконструированного типа этот класс получается путем замены каждого *параметра-типа* в объявлении базового класса соответствующим ему *аргументом-типом* в сконструированном типе. Рассмотрим следующие объявления обобщенных классов:

```
class B<U,V> {…}
class G<T>: B<string,T[]> {…}
```

Здесь базовым классом сконструированного типа `G<int>` является `B<string,int[]>`.

Непосредственный базовый класс класса должен иметь не более строгий вид доступа, чем сам класс (раздел 3.5.2). Например, если `public` класс наследуется от `private` или `internal` класса, возникает ошибка компиляции.

Непосредственный базовый класс класса не может быть одним из следующих типов: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum` или `System.ValueType`. Кроме того, в качестве непосредственного или косвенного базового класса в объявлении обобщенного класса не может использоваться класс `System.Attribute`.

При определении значения спецификации базового класса `A` для класса `B` временно предполагается, что непосредственным базовым классом `B` является `object`. Интуитивно понятно, что это гарантирует, что значение спецификации базового класса не может рекурсивно зависеть от самого себя. Пример:

```
class A<T> {
    public class B{}
}

class C : A<C.B> {}
```

Этот пример ошибочен, так как в спецификации базового класса `A<C.B>` предполагается, что непосредственным базовым классом `C` является `object`; следовательно (в соответствии с правилами из раздела 3.8), не предполагается, что `C` содержит элемент `B`.

К базовым классам класса относятся его непосредственный базовый класс, а также базовые классы последнего. Иными словами, набор базовых классов представляет собой транзитивное замыкание отношения «является непосредственным базовым классом». В приведенном выше примере базовыми классами `B` являются `A` и `object`. Пример:

```
class A {...}

class B<T>: A {...}
```

```
class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

Здесь базовыми классам `D<int>` являются `C<int[]>`, `B<IComparable<int[]>>`, `A` и `object`.

За исключением класса `object`, каждый класс имеет ровно один непосредственный базовый класс. Класс `object` не имеет непосредственного базового класса и является основным базовым классом всех остальных классов.

Когда класс `B` наследуется от класса `A` и при этом `A` зависит от `B`, возникает ошибка компиляции. Класс **непосредственно зависит от** своего непосредственного базового класса (если он есть) и **непосредственно зависит от** класса, в который он непосредственно вложен (если такой класс есть). С учетом этого определения, полный набор классов, от которых зависит класс, представляет собой рефлексивное транзитивное замыкание отношения «**непосредственно зависит от**».

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Это правило игнорирует аргументы-типы, если они присутствуют. Например, несмотря на то что `A<T>` и `A<A<T>>` являются разными типами, следующее объявление все равно некорректно:
>
> ```
> class A<T> : A<A<T>> { }
> ```
>
> И наоборот, использование класса в качестве аргумента-типа для сконструированного типа, указанного в качестве его базового класса, является совершенно корректным:
>
> ```
> class A<T> { }
> class B : A<B[]> { } // Все в порядке
> ```

Пример:

```
class A: A {}
```

является ошибочным, поскольку класс зависит от самого себя. Аналогично, пример

```
class A: B {}

class B: C {}

class C: A {}
```

также ошибочен, поскольку классы циклически зависят от самих себя. И наконец, пример

```
class A: B.C {}

class B: A
{
    public class C {}
}
```

приводит к ошибке компиляции, так как `A` зависит от `B.C` (являющегося его непосредственным базовым классом), который зависит от `B` (являющегося его непосредственным охватывающим классом), который циклически зависит от `A`.

Заметьте, что класс не зависит от вложенных в него классов. Пример:

```
class A
{
    class B: A {}
}
```

Здесь `B` зависит от `A` (поскольку `A` является одновременно и его непосредственным базовым классом, и непосредственным охватывающим классом), но `A` не зависит от `B` (поскольку `B` не является ни базовым, ни охватывающим классом для `A`). Следовательно, пример корректен.

Невозможно наследовать от класса с модификатором `sealed`. Пример:

```
sealed class A {}
class B: A {}        // Ошибка: нельзя наследовать от бесплодного класса
```

Здесь класс `B` является ошибочным, так как он пытается наследовать от бесплодного класса `A`.

10.1.4.2. Реализации интерфейсов

Спецификация *базового-класса* может содержать список интерфейсов. В этом случае говорится, что класс непосредственно реализует указанные интерфейсы. Реализации интерфейсов рассматриваются подробнее в разделе 13.4.

10.1.5. Ограничения на параметры-типы

В объявлениях обобщенных типов и методов можно указывать необязательные ограничения на параметры-типы путем добавления *ограничений-на-параметры-типы*:

ограничения-на-параметры-типы:
  ограничение-на-параметр-тип
  ограничения-на-параметры-типы   *ограничение-на-параметр-тип*

ограничение-на-параметр-тип:
  where   *параметр-тип*   **:**   *список-ограничений-на-параметры-типы*

список-ограничений-на-параметры-типы:
  первичное-ограничение
  вторичное-ограничение
  ограничение-конструктора
  первичное-ограничение   **,**   *вторичные-ограничения*
  первичное-ограничение   **,**   *ограничение-конструктора*
  вторичные-ограничения   **,**   *ограничение-конструктора*
  первичное-ограничение   **,**   *вторичные-ограничения*   **,**   *ограничение-конструктора*

первичное-ограничение:
  класс
  class
  struct

вторичные-ограничения:
  интерфейс
  параметр-тип
  вторичные-ограничения   **,**   *интерфейс*

```
   вторичные-ограничения   ,   параметр-тип
ограничение-конструктора:
   new  (  )
```

Каждое *ограничение-на-параметры-типы* состоит из лексемы `where`, за которой следуют имя параметра-типа, запятая и список ограничений для этого параметра-типа. Для каждого параметра-типа может присутствовать не более одного условия `where`, и эти условия могут быть перечислены в любом порядке. Так же как и лексемы `get` и `set` в коде доступа свойства, лексема `where` не является ключевым словом.

Список ограничений в условии `where` может содержать любые из следующих компонентов в следующем порядке: одно первичное ограничение, одно или несколько вторичных ограничений, ограничение конструктора — `new()`.

Первичное ограничение может быть классом, **ограничением ссылочного типа** `class` или **ограничением типа-значения** `struct`. Вторичное ограничение может быть *параметром-типом* или *интерфейсом*.

Ограничение ссылочного типа указывает на то, что аргумент-тип, используемый в качестве параметра-типа, должен являться ссылочным типом. Все классы, интерфейсы, делегаты, массивы, а также параметры-типы, являющиеся ссылочными типами (в соответствии с приведенным ниже определением), удовлетворяют этому ограничению.

Ограничение типа-значения указывает на то, что аргумент-тип, используемый в качестве параметра-типа, должен являться необнуляемым типом-значением. Все необнуляемые структурные и перечислимые типы, а также параметры-типы, имеющие ограничение типа-значения, удовлетворяют этому ограничению. Несмотря на то что необнуляемый тип классифицируется как тип-значение (раздел 4.1.10), он не удовлетворяет ограничению типа-значения. Параметр-тип, имеющий ограничение типа-значения, не может одновременно содержать *ограничение-конструктора*.

БИЛЛ ВАГНЕР

В какой-то момент этот набор ограничений казался мне излишне ограничивающим. Я хотел заниматься метапрограммированием с использованием обобщений, имея возможность указывать в качестве ограничений любое произвольное множество элементов, таких как сигнатуры других конструкторов или операции. Теперь, когда в C# 3.0 появились лямбда-выражения и более естественная поддержка использования методов в качестве параметров, это ощущение исчезло. Указывая сигнатуры делегатов в качестве параметров-типов или параметров методов, программисты на C# могут достичь практически любых целей.

Типы указателей не могут быть аргументами-типами и не удовлетворяют ни ограничению ссылочного типа, ни ограничению типа-значения.

Если ограничением является класс, интерфейс или параметр-тип, этот тип определяет минимальный «базовый тип», который должен поддерживаться каждым аргументом-типом, используемым в качестве параметра-типа. При использовании сконструированного типа или обобщенного метода аргумент-тип проверяется на соответствие ограничениям параметра-типа во время компиляции. Переданный аргумент-тип должен удовлетворять условиям, описанным в разделе 4.4.4.

Ограничение *класса* должно подчиняться следующим правилам:

- Тип должен являться классом.
- Тип не должен быть бесплодным.
- Тип не должен быть `System.Array`, `System.Delegate`, `System.Enum` и `System.ValueType`.

> **ДЖОН СКИТ**
>
> Я не вижу причины запрещать здесь использование в качестве типа `System.Enum` или `System.Delegate`, хотя, безусловно, более приятной была бы возможность писать «`where T : enum`», а не «`where T : struct, Enum`», при том что последнее использовалось бы наиболее часто. Хотя спецификация C# отделена от CLI, вы можете заподозрить, что данное ограничение продиктовано CLI, но это не так. На самом деле, в спецификации ECMA-335 явно перечислены это и другие ограничения, запрещенные в C#. Возможно, этот запрет будет убран в будущих версиях языка; существует множество ситуаций, в которых его отсутствие было бы полезным.

- Тип не должен являться типом `object`. Так как все типы наследуются от `object`, такое ограничение не оказывало бы никакого влияния, даже если было бы разрешено.
- Для заданного параметра-типа допускается наличие не более одного ограничения класса.

Тип, указанный в качестве ограничения *интерфейса*, должен подчиняться следующим правилам:

- Тип должен являться интерфейсом.
- Тип не может быть указан более одного раза в данном условии `where`.

В любом случае, ограничение может задействовать любые параметры-типы, расположенные в связанном с ним объявлении типа или метода, как часть сконструированного типа, а также содержать объявляемый тип.

Любой класс или интерфейс, указанный в качестве ограничения на параметр-тип, должен иметь не более строгий вид доступа (раздел 3.5.4), чем объявляемый обобщенный тип или метод.

Тип, указанный в качестве ограничения *параметра-типа*, должен подчиняться следующим правилам:

- Тип должен являться параметром-типом.
- Тип не может быть указан более одного раза в данном условии `where`.

Кроме этого, в графе зависимостей параметров-типов не должно быть циклов. Здесь зависимость — это транзитивное отношение, определяемое следующим образом:

- Если параметр-тип `T` используется в качестве ограничения на параметр-тип `S`, то `S` **зависит от** `T`.
- Если параметр-тип `S` зависит от параметра-типа `T` и `T` зависит от параметра-типа `U`, то `S` **зависит от** `U`.

С учетом этого отношения зависимость (прямая или косвенная) параметра-типа от самого себя приводит к ошибке компиляции.

Любые ограничения должны быть согласованы с зависимыми параметрами-типами. Если параметр-тип `S` зависит от параметра-типа `T`, то:

- `T` не должен иметь ограничение типа-значения. В противном случае `T` в сущности является бесплодным, следовательно, можно ограничить `S` таким же типом, что и `T`, убрав тем самым необходимость использования двух параметров-типов.
- Если `S` имеет ограничение типа-значения, `T` не должен иметь ограничение *класса*.
- Если `S` имеет ограничение *класса* `A` и `T` имеет ограничение *класса* `B`, должно существовать тождественное преобразование или неявное ссылочное преобразование из `A` в `B`, или неявное ссылочное преобразование из `B` в `A`.
- Если `S` также зависит от параметра-типа `U` и `U` имеет ограничение *класса* `A`, а `T` имеет ограничение *класса* `B`, должно существовать тождественное преобразование или неявное ссылочное преобразование из `A` к `B`, или неявное ссылочное преобразование из `B` к `A`.

Ситуация, в которой `S` имеет ограничения типа-значения, а `T` — ограничение ссылочного типа, является корректной. В сущности, это ограничивает `T` типами `System.Object`, `System.ValueType`, `System.Enum` и любым типом интерфейса.

Если условие `where` для параметра-типа содержит ограничение конструктора (имеющее форму `new()`), можно использовать операцию `new` для создания экземпляров типа (раздел 7.6.10.1). Любой аргумент-тип, использованный в качестве параметра-типа с ограничением конструктора, должен содержать открытый конструктор без параметров (такой конструктор неявно существует у каждого типа-значения) или являться параметром-типом, содержащим ограничение типа-значения или ограничение конструктора (подробности см. в разделе 10.1.5).

Ниже приведены примеры ограничений:

```
interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}
```

продолжение ➻

```
class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}
```

Следующий пример является ошибочным, так как он приводит к появлению цикла в графе зависимостей параметров-типов:

```
class Circular<S,T>
    where S: T
    where T: S        // Ошибка: цикл в графе зависимостей
{
    ...
}
```

Следующий пример демонстрирует еще несколько ошибочных ситуаций:

```
class Sealed<S,T>
   where S: T
   where T: struct    // Ошибка: T является бесплодным
{
    ...
}

class A {...}

class B {...}

class Incompat<S,T>
   where S: A, T
   where T: B        // Ошибка: несовместимые ограничения класса
{
    ...
}

class StructWithClass<S,T,U>
   where S: struct, T
   where T: U
   where U: A        // Ошибка: A несовместим со struct
{
    ...
}
```

Эффективный базовый класс параметра-типа `T` определяется следующим образом:

- Если у `T` нет первичных ограничений или ограничений на параметры-типы, его эффективным базовым классом является `object`.
- Если `T` имеет ограничение типа-значения, его эффективным базовым классом является `System.ValueType`.
- Если `T` имеет ограничение *класса* `C`, но не имеет ограничений *параметра-типа*, его эффективным базовым классом является `C`.
- Если у `T` нет ограничения *класса*, но имеются одно или несколько ограничений *параметра-типа*, его эффективным базовым классом является наиболее охва-

ченный (encompassed) тип (раздел 6.4.3) из множества эффективных базовых классов его ограничений *параметра-типа*. Правила согласованности гарантируют, что такой наиболее охваченный тип существует.

- Если `T` имеет одновременно ограничение *класса* и одно или несколько ограничений *параметра-типа*, его эффективным базовым классом является наиболее охваченный тип (раздел 6.4.3) из множества, состоящего из ограничения *класса* для `T` и эффективных базовых классов его ограничений *параметра-типа*. Правила согласованности гарантируют, что такой наиболее охваченный тип существует.
- Если T имеет ограничение ссылочного типа, но не имеет ограничений *класса*, его эффективным базовым классом является `object`.

Для выполнения этих правил в случае, когда `T` имеет ограничение `V` с *типом-значением*, используйте вместо него наиболее конкретный базовый класс `V`, имеющий тип *класса*. Такая ситуация никогда не возникнет для явно заданного ограничения, но может произойти, если ограничения для обобщенного метода неявно унаследованы объявлением переопределяющего его метода или явной реализацией метода интерфейса.

Эти правила гарантируют, что эффективным базовым классом всегда является *класс*.

ЭРИК ЛИППЕРТ

Предположим, к примеру, что у вас есть следующий код:

```
class B<T> { public virtual void M<U>() where U : T {} }
class D : B<DateTime> { public override void M<V>() }
```

В этом случае эффективным базовым классом `V` является класс `System.ValueType`, а не структура `DateTime`. Аналогично, если бы вместо `DateTime` мы использовали `DateTime[]`, эффективным базовым классом был бы класс `System.Array`, а не массив `DateTime[]`.

Множество эффективных интерфейсов параметра-типа `T` определяется следующим образом:

- Если у `T` нет *вторичных-ограничений*, его множество эффективных интерфейсов пусто.
- Если `T` имеет ограничения *интерфейса*, но не имеет ограничений *параметра-типа*, его множеством эффективных интерфейсов является множество ограничений *интерфейса*.
- Если у `T` нет ограничений *интерфейса*, но имеются ограничения *параметра-типа*, множеством эффективных интерфейсов является объединение множеств эффективных интерфейсов его ограничений *параметра-типа*.
- Если `T` имеет одновременно ограничения *интерфейса* и ограничения *параметра-типа*, множеством эффективных интерфейсов является объединение его множества ограничений *интерфейса* с множествами эффективных интерфейсов его ограничений *параметра-типа*.

Параметр-тип **заведомо является ссылочным типом**, если он имеет ограничение ссылочного типа или его эффективный базовый класс отличен от `object` и `System.ValueType`.

Значения параметра-типа ограниченного типа могут использоваться для доступа к элементам экземпляра, подразумеваемым ограничениями. Пример:

```
interface IPrintable
{
    void Print();
}
class Printer<T> where T: IPrintable
{
   void PrintOne(T x) {
       x.Print();
   }
}
```

Здесь методы `IPrintable` могут быть непосредственно вызваны через `x`, так как из ограничения для `T` следует, что он всегда реализует `IPrintable`.

10.1.6. Тело класса

Тело-класса определяет элементы этого класса.

```
тело-класса:
   {   объявления-элементов-классаopt   }
```

10.2. Частичные типы

Объявление типа может быть разделено между несколькими **объявлениями частичных типов**. Объявление типа составляется из его частей в соответствии с правилами, приведенными в этом разделе, поэтому на последних этапах компиляции и во время выполнения оно рассматривается как единое объявление.

Объявление-класса, *объявление-структуры* и *объявление-интерфейса* является объявлением частичного типа, если оно содержит модификатор `partial`. Заметьте, что `partial` не является ключевым словом и действует как модификатор, только если располагается непосредственно перед одним из ключевых слов `class`, `struct` или `interface` в объявлении типа или перед типом `void` в объявлении метода. В других контекстах это слово может использоваться как обычный идентификатор.

Каждая часть объявления частичного типа должна содержать модификатор `partial`. Она должна иметь такое же имя и быть объявлена в том же пространстве имен или объявлении типа, что и другие части. Модификатор `partial` означает, что где-то могут существовать дополнительные части объявления типа, но их существование не требуется; модификатор `partial` может содержаться и в типе, имеющем единственное объявление.

Все части частичного типа должны компилироваться совместно так, чтобы на этапе компиляции они могли быть объединены в одно объявление типа. С помощью частичных типов не допускается расширение уже скомпилированных типов.

Объявления вложенных типов могут разделяться на несколько частей с помощью модификатора `partial`. Обычно охватывающий тип также объявляется с использованием этого модификатора, и все части вложенного типа объявляются в разных частях охватывающего типа.

Не допускается использование модификатора `partial` в объявлениях делегатов и перечислений.

БИЛЛ ВАГНЕР

Эта возможность, несомненно, была добавлена для поддержки генераторов кода, но существуют и другие варианты ее использования. Я размещал вложенные классы в разных единицах компиляции и разбивал классы на основе других логических границ. Однако в общем случае не рекомендуется разделять классы только для того, чтобы несколько разработчиков могли работать над одним классом.

БРЭД АБРАМС

Традиционная модель, используемая средствами визуального проектирования программ, заключается в предоставлении разработчикам некоторого пользовательского интерфейса, позволяющего им реализовывать свои замыслы; после этого средство генерирует исходный код, основываясь на этих замыслах. Такой подход проверен временем и широко используется. Эта базовая модель используется, к примеру, при проектировании данных, в ASP.NET и WinForms. Однако данная модель содержит несколько проблем — а именно, разработчикам часто требуется настраивать, модифицировать или расширять код, сгенерированный программой. Популярным решением является непосредственное редактирование сгенерированного кода, но его главный недостаток заключается в том, что оно делает средство визуального проектирования непригодным. Другой подход заключается в наследовании от сгенерированного кода, но он часто усложняется проблемами с типами. Частичные типы и методы дают возможность частично генерировать класс средствами проектирования и частично модифицировать его вручную, что лучше соответствует описанному сценарию.

10.2.1. Атрибуты

Атрибуты частичного типа определяются путем соединения в произвольном порядке атрибутов каждой из частей. Размещение атрибута в нескольких частях эквивалентно указанию его в типе несколько раз. Рассмотрим две части:

```
[Attr1, Attr2("привет")]
partial class A {}

[Attr3, Attr2("пока")]
partial class A {}
```

Они эквивалентны следующему объявлению:

```
[Attr1, Attr2("привет"), Attr3, Attr2("пока")]
class A {}
```

Схожим образом можно объединять атрибуты для параметров-типов.

10.2.2. Модификаторы

Когда объявление частичного типа содержит спецификацию вида доступа (модификаторы `public`, `protected`, `internal` и `private`), она должна согласовываться со всеми остальными частями, содержащими такую спецификацию. Если спецификация вида доступа не содержится ни в одной из частей, для типа используется соответствующий вид доступа по умолчанию (раздел 3.5.1).

Если одно или несколько частичных объявлений вложенного типа содержат модификатор `new` и при этом вложенный тип скрывает унаследованный элемент (раздел 3.7.1.2), никаких предупреждений не возникает.

Если одно или несколько частичных объявлений класса содержат модификатор `abstract`, класс считается абстрактным (раздел 10.1.1.1). В противном случае класс считается неабстрактным.

Если одно или несколько частичных объявлений класса содержат модификатор `sealed`, класс считается бесплодным (раздел 10.1.1.2). В противном случае считается, что от класса можно наследовать.

Заметьте, что класс не может одновременно быть абстрактным и бесплодным.

Когда в объявлении частичного типа используется модификатор `unsafe`, небезопасным контекстом (раздел 18.1) считается только одна конкретная часть.

10.2.3. Параметры-типы и ограничения

Если объявление обобщенного типа разбито на несколько частей, параметры-типы должны быть указаны в каждой части. Каждая часть должна иметь одинаковое число параметров-типов и одинаковые имена параметров в том же порядке следования.

Когда объявление частичного обобщенного типа содержит ограничения (условия `where`), они должны согласовываться со всеми остальными частями, содержащими ограничения. Точнее, ограничения, содержащиеся в каждой части, должны накладываться на одно и то же множество параметров-типов, и для каждого параметра-типа множества первичных и вторичных ограничений и ограничений конструктора должны быть эквивалентны. Два множества ограничений считаются эквивалентными, если они содержат одинаковые элементы. Если ни одна из частей частичного обобщенного типа не содержит ограничений на параметры-типы, последние считаются неограниченными.

Пример:

```
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
```

```
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

Этот пример корректен, поскольку части, содержащие ограничения (первые две), фактически указывают одинаковые множества первичных и вторичных ограничений и ограничений конструктора соответственно для одинаковых множеств параметров-типов.

> **БИЛЛ ВАГНЕР**
>
> Я предпочитаю по возможности указывать параметры-типы и ограничения для всех частей. Такой подход повышает читабельность кода.

10.2.4. Базовый класс

Когда объявление частичного класса содержит спецификацию базового класса, она должна согласовываться со всеми остальными частями, содержащими спецификацию базового класса. Если ни одна из частей частичного класса не содержит спецификацию базового класса, этим классом становится `System.Object` (раздел 10.1.4.1).

10.2.5. Базовые интерфейсы

Множество базовых интерфейсов для типа, объявление которого разделено на несколько частей, представляет собой объединение базовых интерфейсов, указанных для каждой части. Имя конкретного базового интерфейса может встречаться в каждой части только один раз, но разные части могут содержать одни и те же базовые интерфейсы. Для каждого базового интерфейса должна существовать только одна реализация его элементов.

Пример:

```
partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}
```

Здесь множество базовых интерфейсов для класса `C` содержит `IA`, `IB` и `IC`.

Обычно каждая часть предоставляет реализацию объявленного(ых) в ней интерфейса(ов), однако это не является требованием. Часть может предоставлять реализацию интерфейса, объявленного в другой части:

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
}

partial class X: IComparable
{
    ...
}
```

10.2.6. Элементы

За исключением частичных методов (раздел 10.2.7) множество элементов типа, объявленных в разных частях, представляет собой простое объединение элементов, объявленных в каждой части. Тела всех частей объявления типа разделяют одну область объявлений (раздел 3.3), а область видимости каждого элемента (раздел 3.7) распространяется на тела всех частей. Область доступа любого элемента всегда включает все части охватывающего типа; элемент с модификатором `private`, объявленный в одной части, свободно доступен из другой. Объявление одного и того же элемента в более чем одной части типа приводит к ошибке компиляции, если только этот элемент не является типом с модификатором `partial`.

БИЛЛ ВАГНЕР

С точки зрения логики вы можете рассматривать следующий пример как один большой исходный файл, за исключением того, что вам неизвестен порядок, в котором соединяется содержимое классов.

```
partial class A
{
   int x;                     // Ошибка: невозможно объявить x
                              // более одного раза

   partial class Inner        // Все в порядке: Inner — частичный тип
   {
       int y;
   }
}

partial class A
{
   int x;                     // Ошибка: невозможно объявить x
                              // более одного раза

   partial class Inner        // Все в порядке: Inner — частичный тип
   {
       int z;
   }
}
```

Порядок элементов внутри типа редко имеет значение в коде на C#, однако он может быть важен при взаимодействии с другими языками и средами. В этих случаях порядок элементов внутри типа, объявление которого состоит из нескольких частей, не определен.

ДЖОН СКИТ

Одной из областей, в которой в C# важен порядок следования, являются инициализаторы статических переменных и переменных экземпляра: гарантируется, что они выполняются в «текстовом порядке» (раздел 10.5.5), в котором следуют в классе. Это не создает больших проблем, так как идея полагаться на такое упорядочивание обычно является плохой: класс, который перестает работать, когда вы просто переставляете объявления, слишком хрупок, чтобы его использовать.

Будет вполне разумно ожидать, что элементы, объявленные *внутри одной части*, обрабатываются в очевидном порядке, хотя спецификация и не гарантирует этого явным образом.

10.2.7. Частичные методы

Частичные методы могут объявляться в одной части объявления типа и реализовываться в другой. Реализация является необязательной; если ни одна из частей не реализует частичный метод, его объявление и все вызовы этого метода удаляются из объявления типа, получающегося в результате соединения частей.

ВЛАДИМИР РЕШЕТНИКОВ

Частичный метод может быть объявлен только в частичном классе или частичной структуре. Его нельзя объявить в типе, который не является частичным, или в интерфейсе.

Для частичных методов не допускается использовать модификаторы доступа, но неявно они являются закрытыми. Их возвращаемым типом должен быть `void`, а параметры не могут содержать модификатор `out`. Идентификатор `partial` рассматривается в объявлении метода как ключевое слово, только если он находится непосредственно перед типом `void`; в противном случае его можно использовать как обычный идентификатор. Частичный метод не может явно реализовывать методы интерфейса.

Существует два типа объявлений частичного метода. Если телом объявления метода является точка с запятой, объявление называется **определяющим объявлением частичного метода**. Если тело метода представляет собой *блок*, объявление называется **реализующим объявлением частичного метода**. Среди частей объявления типа может присутствовать только одно определяющее объявление частичного метода с конкретной сигнатурой и только одно реализующее объявление частичного метода с конкретной сигнатурой. Если существует реализующее объявление частичного метода, должно также существовать соответствующее определяющее

объявление частичного метода, и объявления должны совпадать в соответствии со следующими правилами:

- Объявления должны иметь одинаковые модификаторы (хотя их порядок может быть разным), имя метода, количество параметров-типов и количество параметров.
- Соответственные параметры в объявлениях должны иметь одинаковые модификаторы (хотя их порядок может быть разным) и одинаковые типы (не считая различий в именах параметров-типов).
- Соответственные параметры-типы в объявлениях должны иметь одинаковые ограничения (не считая различий в именах параметров-типов).

Реализующее объявление частичного метода может находиться в той же части, что и соответствующее определяющее объявление частичного метода.

В разрешении перегрузки принимает участие только определяющий частичный метод. Таким образом, вне зависимости от того, присутствует ли реализующее объявление, выражения вызова могут разрешаться в вызовы частичного метода. Так как частичный метод всегда возвращает `void`, такие выражения вызова всегда будут являться операторами-выражениями. Более того, поскольку частичный метод неявно является закрытым, эти операторы всегда будут находиться в одной из тех частей объявления типа, в которой объявлен частичный метод.

Если ни одна из частей объявления частичного типа не содержит реализующего объявления заданного частичного метода, любой оператор-выражение, вызывающий этот метод, просто удаляется из объединенного объявления типа. Таким образом, выражение вызова, включая любые составляющие его выражения, не оказывает никакого действия во время выполнения. Сам частичный метод также удаляется и не будет являться элементом объединенного объявления типа.

Если для заданного частичного метода существует реализующее объявление, вызовы этого метода сохраняются. Частичный метод приводит к созданию объявления метода, похожего на реализующее объявление частичного метода, за исключением следующего:

- Оно не содержит модификатор `partial`.
- Атрибуты результирующего объявления метода представляют собой объединение атрибутов определяющего и реализующего объявлений частичного метода, при этом их порядок не определен. Дубликаты не удаляются.
- Атрибуты для параметров результирующего объявления метода представляют собой объединение атрибутов соответствующих параметров определяющего и реализующего объявлений частичного метода, при этом их порядок не определен. Дубликаты не удаляются.

Если для частичного метода `M` существует определяющее объявление, но не существует реализующего, применяются следующие ограничения:

- Создание делегата для метода (раздел 7.6.10.5) приводит к ошибке компиляции.
- Попытка сослаться на `M` внутри анонимной функции, которая преобразуется в тип дерева выражений (раздел 6.5.2), приводит к ошибке компиляции.

- Выражения, встречающиеся как часть вызова `M`, не влияют на состояние явного присваивания (раздел 5.3), что потенциально может привести к ошибкам компиляции.
- `M` не может являться точкой входа приложения (раздел 3.1).

Частичные методы полезны, если требуется разрешить одной части объявления типа изменять поведение другой части, например той, которая генерируется программно. Рассмотрим следующее объявление частичного класса:

```
partial class Customer
{
   string name;

   public string Name {
         get { return name; }

      set {
         OnNameChanging(value);
         name = value;
         OnNameChanged();
      }
   }

   partial void OnNameChanging(string newName);

   partial void OnNameChanged();
}
```

Если скомпилировать этот класс без каких-либо других частей, определяющие объявления частичных методов и их вызовы будут удалены, и результирующее объединенное объявление класса будет эквивалентно следующему:

```
class Customer
{
   string name;

   public string Name {

      get { return name; }

      set { name = value; }
   }
}
```

Предположим, однако, что существует другая часть, предоставляющая реализующие объявления частичных методов:

```
partial class Customer
{
   partial void OnNameChanging(string newName)
   {
       Console.WriteLine(name + " изменяется на " + newName);
   }
   partial void OnNameChanged()
   {
       Console.WriteLine("Изменено на " + name);
   }
}
```

В этом случае результирующее объединенное объявление класса будет эквивалентно следующему:

```
class Customer
{
   string name;

   public string Name {

      get { return name; }

      set {
         OnNameChanging(value);
         name = value;
         OnNameChanged();
      }
   }

   void OnNameChanging(string newName)
   {
       Console.WriteLine(name + " изменяется на " + newName);
   }

   void OnNameChanged()
   {
       Console.WriteLine("Изменено на " + name);
   }
}
```

КРИС СЕЛЛЗ

До появления частичных методов существовал шаблон использования виртуальных методов следующим образом:

```
        class Base {
              public void Foo() { HelpWithFoo(); ... }
                  // Скомпонован динамически
              protected virtual void HelpWithFoo() {}
                  // В базовом классе ничего не делает
          }
          class Derived : Base {
              protected override void HelpWithFoo() { ... }
          }
```

Этот подход требовал, чтобы в автоматически сгенерированном коде использовались менее эффективные динамически скомпонованные виртуальные методы для «переноса вызовов» в другую часть частичного класса, несмотря на то что реализация была доступна на этапе компиляции. С использованием частичных методов этот шаблон стал более эффективным:

```
partial class MyClass {        // Сгенерирован программно
         public void Foo() { HelpWithFoo(); ... }
             // Скомпонован статически
         partial void HelpWithFoo();
             // Объявляем частичный метод
      }
```

```
        partial class MyClass {        // Создан человеком
      partial void HelpWithFoo() { ... }
      // Реализуем частичный метод
  }
```

ЭРИК ЛИППЕРТ

Мысль Криса ясна. Я бы добавил, что необходимо рассмотреть и другие затраты, кроме лишней пары наносекунд, требующихся для осуществления виртуального вызова. Предположим, что у вас есть автоматически сгенерированный класс, содержащий сотни точек расширения, в которых автоматически созданному коду потребуется вызывать вспомогательные методы, определенные пользователем. Если этот пользователь захочет реализовать только один метод из возможных сотен, весь код для вызовов и все метаданные для методов все равно генерируются. Частичные методы полностью придерживаются принципа «плати за игру»: дополнительный код создается только для тех точек расширения, которые вы на самом деле используете.

10.2.8. Привязка имен

Хотя каждая часть расширяемого типа должна объявляться в одном пространстве имен, части обычно размещают внутри разных объявлений этого пространства. Соответственно, для каждой части могут использоваться различные директивы `using` (раздел 9.4). При интерпретации простых имен (раздел 7.5.2) в пределах одной части рассматриваются только директивы `using` для объявлений(я) пространств(а) имен, охватывающих(его) данную часть. Это может привести к тому, что один и тот же идентификатор будет иметь разные значения в разных частях:

```
namespace N
{
   using List = System.Collections.ArrayList;

   partial class A
   {
      List x;        // x имеет тип System.Collections.ArrayList
   }
}

namespace N
{
   using List = Widgets.LinkedList;

   partial class A
   {
      List y;        // y имеет тип Widgets.LinkedList
   }
}
```

10.3. Элементы класса

Элементы класса включают в себя элементы, введенные соответствующими *объявлениями-элементов-класса*, и элементы, унаследованные от непосредственного базового класса.

```
объявления-элементов-класса:
   объявление-элемента-класса
   объявления-элементов-класса    объявление-элемента-класса

объявление-элемента-класса:
   объявление-константы
   объявление-поля
   объявление-метода
   объявление-свойства
   объявление-события
   объявление-индексатора
   объявление-операции
   объявление-конструктора
   объявление-деструктора
   объявление-статического-конструктора
   объявление-типа
```

Элементы класса делятся на следующие категории:

- Константы, представляющие константные значения, связанные с классом (раздел 10.4).
- Поля, являющиеся переменными класса (раздел 10.5).
- Методы, реализующие вычисления и действия, которые может производить класс (раздел 10.6).
- Свойства, определяющие именованные характеристики и действия, связанные с чтением и записью этих характеристик (раздел 10.7).
- События, определяющие уведомления, которые может генерировать класс (раздел 10.8).
- Индексаторы, позволяющие выполнять доступ к элементам класса по индексу так же (синтаксически), как к массивам (раздел 10.9).
- Операции, определяющие используемые в выражениях операции, которые можно применять к экземплярам класса (раздел 10.10).
- Конструкторы экземпляра, реализующие действия, необходимые для инициализации экземпляров класса (раздел 10.11).
- Деструкторы, реализующие действия, выполняемые перед полным уничтожением экземпляров класса (раздел 10.13).
- Статические конструкторы, реализующие действия, необходимые для инициализации самого класса (раздел 10.12).
- Типы, представляющие собой типы, локальные для класса (раздел 10.3.8).

Все элементы, которые могут содержать исполняемый код, называются *функциональными элементами* класса. К ним относятся методы, свойства, события,

индексаторы, операции, конструкторы экземпляра, деструкторы и статические конструкторы этого класса.

Объявление-класса создает новую область объявлений (раздел 3.3.), а *объявления-элементов-класса*, непосредственно содержащиеся в *объявлении-класса*, вводят в эту область новые элементы. *Объявления-элементов-класса* подчиняются следующим правилам:

- Конструкторы экземпляра, деструкторы и статические конструкторы должны иметь такое же имя, что и непосредственный охватывающий класс. Имена всех остальных элементов должны отличаться от имени этого класса.
- Имена констант, полей, свойств, событий и типов должны отличаться от имен всех остальных элементов, объявленных в том же классе.
- Имя метода должно отличаться от имен всех остальных элементов, не являющихся методами и объявленных в том же классе. Кроме этого, сигнатура (раздел 3.6) метода должна отличаться от сигнатур всех остальных методов, объявленных в том же классе, а сигнатуры двух методов, объявленных в одном классе, не могут отличаться только лишь модификаторами `ref` и `out`.
- Сигнатура конструктора экземпляра должна отличаться от сигнатур всех остальных конструкторов экземпляра, объявленных в том же классе, а сигнатуры двух конструкторов, объявленных в одном классе, не могут отличаться только лишь модификаторами `ref` и `out`.
- Сигнатура индексатора должна отличаться от сигнатур всех остальных индексаторов, объявленных в том же классе.
- Сигнатура операции должна отличаться от сигнатур всех остальных операций, объявленных в том же классе.

Унаследованные элементы в типе класса (раздел 10.3.3) не являются частью области объявлений класса. Таким образом, в производном классе допускается объявлять элемент с тем же именем или сигнатурой, что и унаследованный элемент (тем самым, в сущности, скрывая его).

10.3.1. Тип экземпляра

У каждого объявления класса существует связанный с ним ограниченный (bound) тип (раздел 4.4.3), называемый **типом экземпляра**. Для объявления обобщенного класса тип экземпляра формируется путем создания сконструированного типа (раздел 4.4) из объявления типа, в котором каждый переданный аргумент-тип является соответствующим параметром-типом. Так как тип экземпляра использует параметры-типы, его можно использовать только там, где эти параметры находятся в области видимости — то есть внутри объявления класса. Для кода, написанного внутри объявления класса, типом экземпляра является тип `this`. Для необобщенных классов типом экземпляра является просто объявляемый класс. В следующем примере показаны несколько объявлений классов и соответствующие типы экземпляров:

```
class A<T>                          // Тип экземпляра: A<T>
{
   class B {}                       // Тип экземпляра: A<T>.B

   class C<U> {}                    // Тип экземпляра: A<T>.C<U>
}

class D {}                          // Тип экземпляра: D
```

10.3.2. Элементы сконструированных типов

Неунаследованные элементы сконструированного типа получаются путем замены каждого *параметра-типа* в объявлении элемента соответствующим *аргументом-типом* для сконструированного типа. Процесс замены основан на семантическом значении объявлений типа и не является простой текстовой подстановкой.

В качестве примера рассмотрим следующее объявление обобщенного класса:

```
class Gen<T,U>
{
   public T[,] a;

   public void G(int i, T t, Gen<U,T> gt) {...}

   public U Prop { get {...} set {...} }

   public int H(double d) {...}
}
```

Здесь сконструированный тип `Gen<int[],IComparable<string>>` содержит следующие элементы:

```
public int[,][] a;

public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}

public IComparable<string> Prop { get {...} set {...} }

public int H(double d) {...}
```

Типом элемента `a` в объявлении обобщенного класса `Gen` является «двумерный массив элементов типа `T`», поэтому типом элемента `a` в приведенном выше сконструированном типе является «двумерный массив одномерных массивов элементов типа `int`» или `int[,][]`.

Внутри функциональных элементов экземпляра тип `this` представляет собой тип экземпляра (раздел 10.3.1) охватывающего объявления.

> **ДЖОН СКИТ**
>
> Использование подстановки для обобщений приводит к очевидным сложным ситуациям. Что будет делать следующий код?

```
public class Puzzle<T> {
   public void Method(int i) {}
   public void Method(T t) {}
}
...
new Puzzle<int>().Method(10);
```

«Подстановка» приводит к тому, что два метода имеют совершенно одинаковые сигнатуры — какой же из них будет вызван в выражении `Method(10)`? Я охотно признаю, что не смог бы ответить на этот вопрос, не поискав в спецификации или не проверив. Я настойчиво рекомендую при любой возможности избегать такой двусмысленности. По собственному опыту могу сказать, что подобная ситуация, скорее всего, возникнет для пары индексаторов, один из которых выполняет поиск элемента по позиции (`int`), а другой по ключу (параметр-тип).

ЭРИК ЛИППЕРТ

В проектном решении для обобщений в C# 2.0 проектировщики языка рассматривали возможность запретить даже *объявление* класса, при конструировании которого *потенциально* могла бы возникать двусмысленность сигнатур, похожая на описанную Джоном. К сожалению, такой запрет сделал бы ошибочными некоторые шаблоны кода, возможность использования которых, очевидно, была бы желательной:

```
class C<T> {
    public C(T t) { ... }
    public C(Stream serializedState) { ... }
}
```

Неужели необходимо делать объявление `C<T>` ошибочным только потому, что кто-то может когда-нибудь написать `C<Stream>`? Это выглядит излишним.

Даже в этом случае создавать типы, которые могут привести к двусмысленным сигнатурам при определенном способе конструирования, а затем конструировать их именно таким способом — чрезвычайно плохая практика программирования. В некоторых надуманных случаях она может привести к поведению CLR, зависящему от реализации.

Все элементы обобщенного класса могут использовать параметры-типы из любого охватывающего класса или непосредственно, или как часть сконструированного типа. Когда конкретный закрытый (closed) сконструированный тип (раздел 4.4.2) используется во время выполнения, каждое использование параметра-типа заменяется фактическим аргументом-типом, переданным в сконструированный тип. Пример:

```
class C<V>
{
   public V f1;
   public C<V> f2 = null;
      public C(V x) {
      this.f1 = x;
      this.f2 = this;
   }
}
```

продолжение ☞

```
class Application
{
   static void Main() {
      C<int> x1 = new C<int>(1);
      Console.WriteLine(x1.f1);            // Выводит 1

      C<double> x2 = new C<double>(3.1415);
      Console.WriteLine(x2.f1);            // Выводит 3.1415
   }
}
```

10.3.3. Наследование

Класс **наследует** все элементы своего непосредственного базового класса. Наследование означает, что класс неявно содержит все элементы типа своего непосредственного базового класса, за исключением конструкторов экземпляра, деструкторов и статических конструкторов базового класса. Ниже приведены некоторые важные аспекты наследования:

- Наследование транзитивно. Если `C` наследуется от `B`, а `B` наследуется от `A`, то `C` наследует как элементы, объявленные в `B`, так и элементы, объявленные в `A`.
- Производный класс *расширяет* свой непосредственный базовый класс. Производный класс может добавлять новые элементы к числу унаследованных, но не может удалить определение унаследованного элемента.
- Конструкторы экземпляра, деструкторы и статические конструкторы не наследуются, но все остальные элементы наследуются вне зависимости от объявлений вида доступа (раздел 3.5). Однако в зависимости от вида доступа унаследованные элементы могут быть недоступны в производном классе.

> **ДЖОН СКИТ**
>
> Время от времени разработчики задаются вопросом, почему не наследуются конструкторы экземпляра. С моей точки зрения только конкретный класс знает, какая информация потребуется ему для создания корректного экземпляра. Например, если бы конструкторы экземпляра наследовались, у всех классов существовали бы конструкторы без параметров (потому что он есть у `object`). Какой бы смысл имело создание нового экземпляра класса `FileStream` без указания имени файла или дескриптора?

- Производный класс может **скрывать** (раздел 3.7.1.2) унаследованные элементы, объявляя новые элементы с такими же именами или сигнатурами. Однако заметьте, что скрытие унаследованного элемента не удаляет его — оно лишь делает элемент недоступным непосредственно через производный класс.
- Экземпляр класса содержит набор всех полей экземпляра, объявленных в классе и его базовых классах, и существует неявное преобразование (раздел 6.1.6) из типа производного класса в любой из типов его базовых классов. Таким образом,

со ссылкой на экземпляр некоторого производного класса можно работать как со ссылкой на экземпляр любого из его базовых классов.

- Класс может объявлять виртуальные методы, свойства и индексаторы, а производные классы могут переопределять реализации этих функциональных элементов. Таким образом, классы могут проявлять полиморфное поведение, при котором действия, выполняемые при вызове функционального элемента, различаются в зависимости от типа времени выполнения экземпляра, через который вызывается данный элемент.

Унаследованные элементы сконструированного класса — это элементы типа непосредственного базового класса (раздел 10.1.4.1), которые определяются путем замены каждого параметра-типа в *спецификации-базового-класса* соответствующим аргументом-типом в сконструированном типе. Эти элементы, в свою очередь, преобразуются путем замены каждого *параметра-типа* в объявлении элемента соответствующим ему *аргументом-типом* в *спецификации-базового-класса*.

```
class B<U>
{
    public U F(long index) {...}
}

class D<T>: B<T[]>
{
    public T G(string s) {...}
}
```

В приведенном примере сконструированный тип `D<int>` содержит неунаследованный элемент `public int G(string s)`, полученный заменой параметра-типа `T` аргументом-типом `int`. `D<int>` также содержит унаследованный элемент из объявления класса `B`. Этот элемент определяется следующим образом: сначала путем замены `T` на `int` в спецификации базового класса `B<T[]>` определяется базовый класс для `D<int>`, имеющий тип `B<int[]>`; затем `U` в объявлении `public U F(long index)` заменяется аргументом-типом для B — `int[]`, — в результате чего получается унаследованный элемент `public int[] F(long index)`.

10.3.4. Модификатор new

Объявление-элемента-класса может объявлять элемент с таким же именем или сигнатурой, что и у унаследованного элемента. Когда это происходит, принято говорить, что элемент производного класса **скрывает** элемент базового класса. Скрытие унаследованного элемента не считается ошибкой, однако заставляет компилятор выдать предупреждение. Чтобы убрать это предупреждение, объявление элемента производного класса может содержать модификатор `new`, означающий, что скрытие унаследованным элементом базового является намеренным. Более подробно этот вопрос рассматривается в разделе 3.7.1.2.

Если модификатор `new` используется в объявлении, которое не скрывает унаследованный элемент, компилятор выдает предупреждение. Его можно убрать, удалив модификатор `new`.

10.3.5. Модификаторы доступа

Объявление-элемента-класса может иметь любой из пяти возможных объявлений вида доступа (раздел 3.5.1): `public`, `protected internal`, `protected`, `internal` или `private`. За исключением сочетания `protected internal`, указание более чем одного модификатора доступа приводит к ошибке компиляции. Если *объявление-элемента-класса* не содержит модификаторов доступа, по умолчанию используется `private`.

10.3.6. Составляющие типы

Типы, используемые в объявлении элемента, называются составляющими типами этого элемента. К возможным составляющим типам относятся типы константы, поля, свойства, события или индексатора, тип возвращаемого методом или операцией значения, а также типы параметров метода, индексатора, операции или конструктора экземпляра.

Составляющие типы элемента должны иметь не более строгий вид доступа, чем сам элемент (раздел 3.5.4).

10.3.7. Статические элементы и элементы экземпляра

Элементы класса могут являться или **статическими элементами** или **элементами экземпляра**. Вообще говоря, полезно рассматривать статические элементы как принадлежащие к классам, а элементы экземпляра как принадлежащие к объектам (экземплярам классов).

Когда объявление поля, метода, свойства, события, операции или конструктора содержит модификатор `static`, оно объявляет статический элемент. Кроме этого, объявление константы или типа объявляет статический элемент неявно. Статические элементы обладают следующими характеристиками:

- Когда к статическому элементу `M` обращаются с помощью *доступа-к-элементу* (раздел 7.6.4) в форме `E.M`, `E` должен обозначать тип, содержащий `M`. Если `E` обозначает экземпляр, возникает ошибка компиляции.
- Статическое поле определяет ровно одну область памяти, общую для всех экземпляров данного закрытого класса. Независимо от того, сколько экземпляров будет создано, всегда существует только одна копия статического поля.
- Статический функциональный элемент (метод, свойство, события, операция или конструктор) не работает с конкретным экземпляром, поэтому попытка сослаться на `this` в таком элементе приведет к ошибке компиляции.

Когда объявление поля, метода, свойства, события, индексатора, конструктора или деструктора не содержит модификатора `static`, оно объявляет элемент экземпляра (его иногда называют нестатическим элементом). Элементы экземпляра обладают следующими характеристиками:

- Когда к элементу экземпляра `M` обращаются с помощью *доступа-к-элементу* (раздел 7.6.4) в форме `E.M`, `E` должен обозначать экземпляр типа, содержащего `M`. Если `E` обозначает тип, возникает ошибка компоновки.
- Каждый экземпляр класса содержит отдельное множество всех полей экземпляра класса.
- Функциональный элемент экземпляра (метод, свойство, индексатор, конструктор экземпляра или деструктор) работает с данным экземпляром класса и может получать доступ к этому экземпляру с помощью `this` (раздел 7.6.7).

Следующий пример демонстрирует правила доступа к статическим элементам и элементам экземпляра:

```
class Test
{

   int x;
   static int y;

   void F() {
       x = 1;               // Все в порядке: то же, что this.x = 1
       y = 1;               // Все в порядке: то же, что Test.y = 1
   }

   static void G() {
      x = 1;                // Ошибка: нельзя получить доступ к this.x
      y = 1;                // Все в порядке: то же, что Test.y = 1
   }

   static void Main() {
      Test t = new Test();
      t.x = 1;              // Все в порядке
      t.y = 1;              // Ошибка: нельзя получить доступ
                            // к статическому элементу через экземпляр
      Test.x = 1;           // Ошибка: нельзя получить доступ
                            // к элементу экземпляра через тип
      Test.y = 1;           // Все в порядке
    }

}
```

Метод `F` показывает, что в функциональном элементе экземпляра *простое-имя* (раздел 7.6.2) может использоваться для доступа как к элементам экземпляра, так и к статическим элементам.

Метод `G` показывает, что в статическом функциональном элементе доступ к элементу экземпляра через *простое-имя* приводит к ошибке компиляции.

Метод `Main` показывает, что в *доступе-к-элементу* (раздел 7.6.4) доступ к элементам экземпляра должен выполняться через экземпляры, а к статическим элементам — через типы.

10.3.8. Вложенные типы

Тип, объявленный внутри объявления класса или структуры, называется **вложенным типом**. Тип, объявленный внутри единицы компиляции или пространства имен, называется **невложенным типом**.

> **ДЖОН СКИТ**
>
> Я слышал фразу «тип верхнего уровня» чаще, чем «невложенный тип» — и я, несомненно, сам распространяю эту нестандартную терминологию. С учетом того, что подавляющее большинство типов являются невложенными, возникает чувство, что используемый для их описания термин должен быть точным, а не обозначающим просто отсутствие вложенности. В любом случае, это мое объяснение, и я одобряю его заимствование другими.

Пример:

```
using System;

class A
{
   class B
   {
      static void F() {
              Console.WriteLine("A.B.F");
      }
   }
}
```

Здесь класс B является вложенным типом, поскольку он объявлен внутри класса A, а класс B является невложенным типом, так как он объявлен внутри единицы компиляции.

> **БРЭД АБРАМС**
>
> Я не большой любитель добавления открыто доступных вложенных типов в библиотеки, предназначенные для многократного использования. Открытость вложенных типов не представляет собой ничего хорошего. И что более важно, у нас уже есть механизм группировки — пространства имен, — который должен использоваться для обозначения взаимосвязанных типов.

10.3.8.1. Полные имена

Полным именем (раздел 3.8.1) для вложенного типа является `S.N`, где `S` — полное имя типа, в котором объявлен `N`.

10.3.8.2. Объявления вида доступа

Невложенные типы могут иметь вид доступа `public` или `internal`, по умолчанию для них используется `internal`. Вложенные типы тоже могут использовать эти

виды доступа, а также еще один или несколько дополнительных, в зависимости от того, является ли содержащий их тип классом или структурой:

- Вложенный тип, объявленный в классе, может иметь любой из пяти видов доступа (`public`, `protected internal`, `protected`, `internal` или `private`) и, как и другие элементы класса, по умолчанию использует `private`.
- Вложенный тип, объявленный в структуре, может иметь любой из трех видов доступа (`public`, `internal` или `private`) и, как и другие элементы структуры, по умолчанию использует `private`.

Пример:

```
public class List
{
   // Закрытая структура данных
   private class Node
   {
      public object Data;
      public Node Next;

      public Node(object data, Node next) {
         this.Data = data;
         this.Next = next;
       }
    }

   private Node first = null;
   private Node last = null;

   // Открытый интерфейс
   public void AddToFront(object o) {...}

   public void AddToBack(object o) {...}

   public object RemoveFromFront() {...}

   public object RemoveFromBack() {...}

   public int Count { get {...} }
}
```

В этом примере объявляется закрытый вложенный класс `Node`.

10.3.8.3. Скрытие

Вложенный тип может скрывать (раздел 3.7.1) элемент из базового типа. В объявлениях вложенных типов допускается использование модификатора `new`, чтобы явно указать на скрытие. Пример:

```
class Base
{
   public static void M() {
       Console.WriteLine("Base.M");
   }
}
```

продолжение ☞

```
class Derived: Base
{
   new public class M
   {
      public static void F() {
              Console.WriteLine("Derived.M.F");
      }
   }
}

class Test
{
   static void Main() {
       Derived.M.F();
   }
}
```

Здесь продемонстрирован вложенный класс `M`, скрывающий определенный в `Base` метод `M`.

10.3.8.4. this-доступ

Между вложенным типом и содержащим его типом не существует никакой особой взаимосвязи с точки зрения *this-доступа* (раздел 7.6.7). А именно, `this` не может использоваться внутри вложенного типа, чтобы ссылаться на элементы экземпляра содержащего его типа. Если вложенному типу требуется доступ к элементам экземпляра содержащего его типа, такой доступ можно предоставить, передав `this` для экземпляра внешнего типа в конструктор вложенного типа в качестве аргумента. Пример:

```
using System;

class C
{
   int i = 123;

   public void F() {
      Nested n = new Nested(this);
      n.G();
   }

   public class Nested
   {
      C this_c;

      public Nested(C c) {
              this_c = c;
       }

          public void G() {
              Console.WriteLine(this_c.i);
          }
   }
}
```

```
class Test
{
   static void Main() {
      C c = new C();
      c.F();
   }
}
```

Приведенный пример демонстрирует данную технику. Экземпляр `C` создает экземпляр `Nested` и передает собственный `this` в конструктор `Nested`, чтобы предоставить ему доступ к элементам экземпляра `C`.

> **ДЖОН СКИТ**
>
> Этот раздел спецификации на первый взгляд может показаться странным: зачем вообще указывать, чего вы *не можете* сделать? С чего бы кому-то ожидать возможность получить в экземпляре вложенного типа экземпляр содержащего его типа? Ну, в Java внутренние классы работают именно так. Чтобы достичь схожего с C# поведения, вам потребуется объявить вложенный тип как `static`, что не имеет ничего общего со статическим классом в C#!

10.3.8.5. Доступ к закрытым и защищенным элементам охватывающего типа

Вложенный тип имеет доступ ко всем элементам, доступным содержащему его типу, включая те элементы охватывающего типа, для которых объявлен вид доступа `private` и `protected`. Пример:

```
using System;

class C
{
   private static void F() {
       Console.WriteLine("C.F");
   }

   public class Nested
   {
      public static void G() {
              F();
      }
   }
}

class Test
{
   static void Main() {
       C.Nested.G();
   }
}
```

Здесь показан класс `C`, содержащий вложенный класс `Nested`. Метод `G`, расположенный внутри `Nested`, вызывает статический метод `F`, определенный в `C` и объявленный как `private`.

ДЖОН СКИТ

И снова это поведение отличается от Java, где внешний тип имеет доступ к закрытым элементам вложенного типа, но не наоборот. Подход, примененный в C#, кажется мне более целесообразным. Он также позволяет использовать некоторые удачные шаблоны, например создание абстрактного класса с закрытым конструктором: все производные классы должны быть вложены в абстрактный. Как это ни странно, основным способом использования такого поведения мне видится эмуляция перечислений из Java.

Вложенный тип также имеет доступ к защищенным элементам, определенным в базовом типе охватывающего типа. Пример:

```
using System;

class Base
{
   protected void F() {
       Console.WriteLine("Base.F");
   }
}

class Derived: Base
{
   public class Nested
   {
      public void G() {
         Derived d = new Derived();
         d.F();            // Все в порядке
      }
   }
}

class Test
{
   static void Main() {
      Derived.Nested n = new Derived.Nested();
      n.G();
   }
}
```

Здесь вложенный класс `Derived.Nested` обращается к защищенному методу `F`, определенному в базовом классе `Derived` — `Base`, — вызывая его через экземпляр `Derived`.

10.3.8.6. Вложенные типы в обобщенных классах

Объявление обобщенного класса может содержать объявления вложенных типов. Внутри этих типов могут использоваться параметры-типы охватывающего класса. Объявление вложенного типа может содержать дополнительные параметры-типы, применяющиеся только ко вложенному типу.

Каждое объявление типа, находящееся внутри объявления обобщенного типа, неявно является объявлением обобщенного типа. При обращении к типу,

вложенному в обобщенный тип, необходимо использовать имя охватывающего сконструированного типа и его аргументы-типы. Однако внутри внешнего класса вложенный тип может использоваться без полного имени; тип экземпляра внешнего класса можно неявно использовать при конструировании вложенного типа. Следующий пример демонстрирует три различных — и корректных — способа ссылаться на сконструированный тип, созданный из `Inner`; первые два из них эквивалентны:

```
class Outer<T>
{
   class Inner<U>
   {
       public static void F(T t, U u) {...}
   }

   static void F(T t) {
      Outer<T>.Inner<string>.F(t, "abc");   // Эти два оператора
      Inner<string>.F(t, "abc");            // приводят к одному результату

      Outer<int>.Inner<string>.F(3, "abc"); // Этот тип отличается

      Outer.Inner<string>.F(t, "abc");      // Ошибка: для Outer требуется
                                            // аргумент-тип
   }
}
```

Хотя это и является плохим стилем программирования, параметр-тип во вложенном типе может скрывать элемент или параметр-тип, объявленный во внешнем типе:

```
class Outer<T>
{
    class Inner<T>                         // Корректно: скрывает T из Outer
    {
        public T t;                        // Ссылается на T из Inner
    }
}
```

10.3.9. Зарезервированные имена элементов

Чтобы упростить реализацию базовой среды выполнения C#, для каждого объявления исходного элемента, являющегося свойством, событием или индексатором, реализация должна резервировать две сигнатуры метода, основанные на виде (kind) объявления элемента, его имени и типе. Если в программе объявлен элемент, сигнатура которого совпадает с одной из зарезервированных сигнатур, возникает ошибка компиляции, даже если в реализации базовой среды выполнения эти сигнатуры не используются.

Зарезервированные имена не вводят новых объявлений; таким образом, они не участвуют в поиске элементов. Однако ассоциированные с объявлением зарезервированные сигнатуры методов участвуют в наследовании (раздел 10.3.3) и могут быть скрыты с помощью модификатора `new` (раздел 10.3.4).

Резервирование этих имен преследует три цели:

- позволяет базовой реализации использовать обычный идентификатор в качестве имени метода для доступа `get` и `set` к возможностям языка C#;
- позволяет другим языкам использовать для взаимодействия обычный идентификатор в качестве имени метода для доступа `get` и `set` к возможностям языка C#;
- помогает гарантировать, что исходный код, принятый одним соответствующим стандарту компилятором, будет принят другим, делая особенности зарезервированных имен элементов согласованными между всеми реализациями C#.

Объявление деструктора (раздел 10.13) также приводит к резервированию сигнатуры (раздел 10.3.9.4).

10.3.9.1. Имена элементов, зарезервированные для свойств

Для свойства `P` (раздел 10.7) типа `T` резервируются следующие сигнатуры:

```
T get_P();
void set_P(T value);
```

Резервируются обе сигнатуры, даже если свойство доступно только для чтения или только для записи.

> **БИЛЛ ВАГНЕР**
>
> Хотя вы и можете писать методы с именами `get_<что-то>` и `set_<что-то>`, когда `<что-то>` не является именем свойства, это не рекомендуется. Избегая таких имен, вы сведете к минимуму возможную необходимость обновления кода в будущем.

Пример:

```
using System;

class A
{
   public int P {
       get { return 123; }
   }
}

class B: A
{
   new public int get_P() {
       return 456;
   }

   new public void set_P(int value) {
   }
}

class Test
{
```

```
   static void Main() {
      B b = new B();
      A a = b;
      Console.WriteLine(a.P);
      Console.WriteLine(b.P);
      Console.WriteLine(b.get_P());
   }
}
```

Здесь в классе `A` определено свойство `P`, доступное только для чтения, и следовательно, резервируются сигнатуры методов `get_P` и `set_P`. Класс `B` наследуется от класса `A` и скрывает обе эти сигнатуры. Приведенный пример выведет следующее:

```
123
123
456
```

10.3.9.2. Имена элементов, зарезервированные для событий

Для события `E` (раздел 10.8), имеющего тип делегата `T`, зарезервированы следующие сигнатуры:

```
void add_E(T handler);
void remove_E(T handler);
```

10.3.9.3. Имена элементов, зарезервированные для индексаторов

Для индексатора (раздел 10.9) типа `T` со списком параметров `L` зарезервированы следующие сигнатуры:

```
T get_Item(L);
void set_Item(L, T value);
```

Резервируются обе сигнатуры, даже если индексатор доступен только для чтения или только для записи. Более того, зарезервированным является также имя элемента `Item`.

10.3.9.4. Имена элементов, зарезервированные для деструкторов

Для класса, содержащего деструктор (раздел 10.13), зарезервирована следующая сигнатура:

```
void Finalize();
```

10.4. Константы

Константа — это элемент класса, представляющий константное значение, то есть значение, которое можно вычислить на этапе компиляции. *Объявление-константы* вводит одну или несколько констант указанного типа.

```
объявление-константы:
   атрибуты_opt   модификаторы-константы_opt   const   тип   описатели-константы   ;

модификаторы-константы:
   модификатор-константы
   модификаторы-константы   модификатор-константы

модификатор-константы:
   new
   public
   protected
   internal
   private

описатели-константы:
   описатель-константы
   описатели-константы   ,   описатель-константы

описатель-константы:
   идентификатор   =   константное-выражение
```

Объявление-константы может содержать набор *атрибутов* (раздел 17), модификатор `new` (раздел 10.3.4) и корректную комбинацию четырех модификаторов доступа (раздел 10.3.5). Атрибуты и модификаторы применяются ко всем элементам, объявленным *объявлением-константы*. Несмотря на то что константы считаются статическими элементами, в *объявлении-константы* не требуется и не допускается использование модификатора `static`. Если один и тот же модификатор встречается в объявлении несколько раз, возникает ошибка компиляции.

Тип в *объявлении-константы* определяет тип элементов, вводимых этим объявлением. За типом следует список *описателей-констант*, каждый из которых вводит новый элемент. *Описатель-константы* состоит из *идентификатора*, определяющего имя элемента, лексемы «`=`» и *константного-выражения* (раздел 7.19), определяющего значение элемента.

Типом, указанным в объявлении константы, должен быть `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, *перечисление* или *ссылочный-тип*. Результатом вычисления *константного-выражения* должно являться значение целевого типа или типа, который может быть преобразован в целевой тип с помощью неявного преобразования (раздел 6.1).

> **ДЖОН СКИТ**
>
> Интересным моментом здесь является включение в этот список типа `decimal`, когда речь идет об общеязыковой инфраструктуре. В CLI все остальные типы имеют литеральные представления, в то время как у `decimal` оно отсутствует. Компилятор Microsoft C# достигает нужного поведения путем создания объявления статического поля, доступного только для чтения, и применения к нему атрибута `DecimalConstantAttribute`. Похожий атрибут существует и для `DateTime`, но в C# константы этого типа недопустимы.

Тип константы должен иметь не более строгий вид доступа, чем сама константа (раздел 3.5.4).

В выражении значение константы можно получить, используя *простое-имя* (раздел 7.6.2) или *доступ-к-элементу* (раздел 7.6.4).

В *константном-выражении* может использоваться сама константа. Таким образом, константу можно использовать в любой конструкции, в которой требуется *константное-выражение*. В качестве примеров таких конструкций можно привести метки `case`, операторы `goto case`, объявления элементов `enum`, атрибуты и другие объявления констант.

Как описано в разделе 7.19, *константное-выражение* — это выражение, которое может быть полностью вычислено на этапе компиляции. Так как единственным способом создать ненулевое значение *ссылочного-типа*, отличного от `string`, является использование операции `new`, а последняя в *константном-выражении* не допускается, единственным возможным значением для констант *ссылочных-типов*, отличных от `string`, является `null`.

Когда требуется дать константному значению символьное имя, но при этом использование типа данного значения в объявлении константы не допускается или значение нельзя вычислить *константным-выражением* на этапе компиляции, вместо него можно использовать поле `readonly` (раздел 10.5.2).

Объявление константы, которое объявляет несколько констант, эквивалентно нескольким объявлениям одной константы с одинаковыми атрибутами, модификаторами и типом. Пример:

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

Это объявление эквивалентно следующему:

```
class A
{
   public const double X = 1.0;
   public const double Y = 2.0;
   public const double Z = 3.0;
}
```

КРИС СЕЛЛЗ

С точки зрения читабельности, несколько объявлений в одной строке, — будь то константы или не константы, статические элементы или элементы экземпляров, поля или локальные переменные, — кажутся мне сложными для понимания. Я предпочитаю размещать только одно объявление в каждой строке, в идеальном случае учитывая принцип локальности обращений.

Константы могут зависеть от других констант в той же программе, если только эти зависимости не носят циклический характер. Компилятор автоматически организовывает вычисление объявлений констант в нужном порядке. Пример:

```
class A
{
   public const int X = B.Z + 1;
   public const int Y = 10;
}
```

продолжение ☞

```
class B
{
    public const int Z = A.Y + 1;
}
```

В этом примере компилятор сначала вычисляет `A.Y`, затем `B.Z` и наконец, `A.X`, создавая значения `10`, `11` и `12`, именно в таком порядке. Объявления констант могут зависеть от констант из других программ, но такие зависимости могут быть только однонаправленными. Для приведенного выше примера, если бы `A` и `B` были объявлены в разных программах, `A.X` могла бы зависеть от `B.Z`, но `B.Z` не могла бы в то же время зависеть от `A.Y`.

10.5. Поля

Поле — это элемент, представляющий переменную, связанную с объектом или классом. *Объявление-поля* вводит одно или несколько полей указанного типа.

```
объявление-поля:
   атрибуты_opt   модификаторы-поля_opt   тип   описатели-переменных   ;

модификаторы-поля:
   модификатор-поля
   модификаторы-поля   модификатор-поля

модификатор-поля:
   new
   public
   protected
   internal
   private
   static
   readonly
   volatile

описатели-переменных:
   описатель-переменной
   описатели-переменных   ,   описатель-переменной

описатель-переменной:
   идентификатор
   идентификатор   =   инициализатор-переменной

инициализатор-переменной:
   выражение
   инициализатор-массива
```

Объявление-поля может содержать набор *атрибутов* (раздел 17), модификатор `new` (раздел 10.3.4), корректную комбинацию четырех модификаторов доступа (раздел 10.3.5) и модификатор `static` (раздел 10.5.1). Кроме этого, *объявление-поля* может включать модификатор `readonly` (раздел 10.5.2) или `volatile` (раздел 10.5.3), но не оба сразу. Атрибуты и модификаторы применяются ко всем элементам, объявленным *объявлением-поля*. Если один и тот же модификатор встречается в объявлении несколько раз, возникает ошибка компиляции.

Тип в *объявлении-поля* определяет тип элементов, вводимых этим объявлением. За типом следует список *описателей-переменных*, каждый из которых вводит новый элемент. *Описатель-переменной* состоит из *идентификатора*, определяющего имя элемента, за которым следуют необязательные лексема «=» и *инициализатор-переменной* (раздел 10.5.5), определяющий начальное значение элемента.

Тип поля должен иметь не более строгий вид доступа, чем само поле (раздел 3.5.4).

В выражении значение поля можно получить, используя *простое-имя* (раздел 7.6.2) или *доступ-к-элементу* (раздел 7.6.4). Изменить значение поля, доступного не только для чтения, можно с помощью *присваивания* (раздел 7.17). Значение поля, доступного не только для чтения, можно как получить, так и изменить с помощью операций постфиксного инкремента и декремента (раздел 7.6.9) и префиксного инкремента и декремента (раздел 7.7.5).

Объявление поля, которое объявляет несколько полей, эквивалентно нескольким объявлениям одного поля с одинаковыми атрибутами, модификаторами и типами. Пример:

```
class A
{
    public static int X = 1, Y, Z = 100;
}
```

Это объявление эквивалентно следующему:

```
class A
{
   public static int X = 1;
   public static int Y;
   public static int Z = 100;
}
```

10.5.1. Статические поля и поля экземпляра

Если объявление поля содержит модификатор `static`, поля, вводимые этим объявлением, являются **статическими полями**. Если модификатор `static` отсутствует, поля, вводимые объявлением, называются *полями экземпляра*. Статические поля и поля экземпляра представляют собой два из нескольких типов переменных (раздел 5), поддерживаемых C#, и временами их называют соответственно **статическими переменными** и **переменными экземпляра**.

Статическое поле не является частью конкретного экземпляра; напротив, оно разделяется между всеми экземплярами закрытого типа (раздел 4.4.2). Вне зависимости от того, сколько создается экземпляров закрытого класса, в связанном с ним домене приложения всегда существует только одна копия статического поля. Пример:

```
class C<V>
{
   static int count = 0;

   public C() {
       count++;
   }
```

продолжение ☞

```
      public static int Count {
       get { return count; }
   }
}

class Application
{
   static void Main() {
      C<int> x1 = new C<int>();
      Console.WriteLine(C<int>.Count);        // Выводит 1

      C<double> x2 = new C<double>();
      Console.WriteLine(C<int>.Count);        // Выводит 1

      C<int> x3 = new C<int>();
      Console.WriteLine(C<int>.Count);        // Выводит 2
   }
}
```

Поле экземпляра принадлежит экземпляру. А именно, каждый экземпляр класса содержит отдельное множество всех полей экземпляра этого класса.

Когда к полю обращаются с помощью *доступа-к-элементу* (раздел 7.6.4) в форме `E.M`, где `M` — статическое поле, `E` должен обозначать тип, содержащий `M`; если `M` является полем экземпляра, `E` должен обозначать экземпляр типа, содержащего `M`.

Более подробно различия между статическими элементами и элементами экземпляра рассматриваются в разделе 10.3.7.

10.5.2. Поля, доступные только для чтения

Когда *объявление-поля* содержит модификатор `readonly`, поля, вводимые этим объявлением, являются **полями, доступными только для чтения**. Непосредственные присваивания таким полям должны либо являться частью объявления, либо находиться в конструкторе экземпляра или статическом конструкторе того же класса (в этих контекстах значение полю `readonly` можно присвоить несколько раз). Точнее говоря, непосредственные присваивания полю, объявленному с модификатором `readonly`, допускаются только в следующих контекстах:

- В *описателе-переменной*, вводящем это поле (путем добавления в объявление *инициализатора-переменной*).
- Для поля экземпляра — в конструкторах экземпляра класса, содержащего объявление поля; для статического поля — в статическом конструкторе класса, содержащего объявление поля. Кроме того, передача поля `readonly` в качестве параметра `ref` или `out` допускается только в этих контекстах.

Попытка присвоить значению полю, доступному только для чтения, или передать его в качестве параметра `ref` или `out` в любом другом контексте приведет к ошибке компиляции.

ВЛАДИМИР РЕШЕТНИКОВ

В конструкторах экземпляра присваивать значения можно только полям `readonly` объекта `this`, но не полям любых других объектов того же типа:

```
class A
{
    readonly int x;

    A(A a)
    {
        this.x = 1;     // Все в порядке
        a.x = 1;        // Ошибка CS0191: Невозможно присвоить
                        // значение полю, доступному только
                        // для чтения
    }
}
```

Кроме того, полю `readonly` нельзя присвоить значение (или передать его в качестве параметра `ref` или `out`) в анонимной функции, даже если эта функция находится внутри конструктора.

10.5.2.1. Использование статических полей, доступных только для чтения, вместо констант

Поле, объявленное как `static readonly`, оказывается полезным, когда требуется указать символьное имя для константного значения, но в объявлении `const` не допускается использовать тип этого значения или значение нельзя вычислить на этапе компиляции.

Пример:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

Здесь элементы `Black`, `White`, `Red`, `Green` и `Blue` не могут быть объявлены как константные элементы, поскольку их значения невозможно вычислить на этапе компиляции. Объявление их вместо этого как `static readonly` приводит к почти такому же результату.

10.5.2.2. Управление версиями констант и статических полей, доступных только для чтения

Константы и поля `readonly` имеют различную семантику управления версиями двоичных файлов. Когда выражение ссылается на константу, значение константы получается на этапе компиляции, но когда выражение ссылается на поле `readonly`, значения поля неизвестно до времени выполнения. Рассмотрим приложение, состоящее из двух отдельных программ:

```
using System;

namespace Program1
{
   public class Utils
   {
       public static readonly int X = 1;
   }
}

namespace Program2
{
   class Test
   {
      static void Main() {
              Console.WriteLine(Program1.Utils.X);
      }
   }
}
```

Пространства имен `Program1` и `Program2` обозначают две программы, компилируемые отдельно. Поскольку поле `Program1.Utils.X` объявлено как `static readonly`, значение, выводимое оператором `Console.WriteLine`, неизвестно на этапе компиляции и получается во время выполнения. Таким образом, если изменить значение `X` и перекомпилировать `Program1`, оператор `Console.WriteLine` выведет новое значение, даже если `Program2` не перекомпилировалась. Однако если бы `X` было константой, его значение определялось бы на этапе компиляции `Program2` и не затрагивалось бы изменениями в `Program1` до момента перекомпиляции `Program2`.

БИЛЛ ВАГНЕР

Это рассуждение объясняет, почему зачастую предпочтительнее использовать `readonly`, а не `const`.

ДЖОН СКИТ

Хотя я по большей части согласен с Биллом в том, что использование `readonly` обычно более предпочтительно, в некоторых ситуациях `const` обладает преимуществом. Выражения, состоящие из констант, могут вычисляться на этапе компиляции, а не при каждом доступе. В случае вычисления строковой константы такая практика также по-

зволяет избежать создания новых строк при каждом вычислении выражения. Некоторые числа *на самом деле* являются константами, например число миллисекунд в секунде или наименьшее значение `int`. Когда абсолютно точно известно, что значение не будет изменяться, имеет смысл использовать `const`. Если есть хотя бы тень сомнения, безопаснее использовать `readonly`.

10.5.3. Асинхронно-изменяемые поля

Когда *объявление-поля* содержит модификатор `volatile`, поля, вводимые этим объявлением, являются **асинхронно-изменяемыми полями**.

Для полей, не являющихся асинхронно-изменяемыми, техники оптимизации, которые переупорядочивают инструкции, могут привести к неожиданным и непредсказуемым результатам в многопоточных программах, выполняющих доступ к полям без синхронизации, предоставляемой, к примеру, *оператором-lock* (раздел 8.12). Эти оптимизации могут выполняться компилятором, средой выполнения или аппаратными средствами. Для асинхронно-изменяемых полей такие оптимизации ограничены следующим образом:

- Чтение асинхронно-изменяемого поля называется **временным чтением** (volatile read). Временное чтение обладает «семантикой владения» («acquire semantics»), то есть гарантируется, что оно будет выполняться перед любыми обращениями к памяти, находящимися после него в последовательности инструкций.
- Запись асинхронно-изменяемого поля называется **временной записью** (volatile write). Временная запись обладает «семантикой освобождения» («release semantics»), то есть гарантируется, что она будет выполняться после любых обращений к памяти, находящихся перед командой записи в последовательности инструкций.

Эти ограничения гарантируют, что временные записи, осуществленные в некотором потоке, будут наблюдаться во всех остальных потоках в том порядке, в котором они совершались. От соответствующей стандарту реализации не требуется обеспечение единого общего порядка временных записей, наблюдаемого из всех выполняющихся потоков. Асинхронно-изменяемое поле должно иметь один из следующих типов:

- *Ссылочный-тип*.
- Тип `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, `bool`, `System.IntPtr` или `System.UIntPtr`.
- *Перечислимый-тип*, имеющий в качестве базового типа перечисления `byte`, `sbyte`, `short`, `ushort`, `int` или `uint`.

Пример:

```
using System;
using System.Threading;
```

продолжение ⇗

```
class Test
{
   public static int result;
   public static volatile bool finished;

   static void Thread2() {
      result = 143;
      finished = true;
   }

   static void Main() {
      finished = false;

      // Запускаем Thread2() в новом потоке
      new Thread(new ThreadStart(Thread2)).Start();
            // Ждем, пока Thread2 не сообщит о наличии результата,
      // установив для finished значение true
      for (;;) {
         if (finished) {
            Console.WriteLine("result = {0}", result);
            return;
         }
      }
   }
}
```

Данная программа выведет следующее:

```
result = 143
```

В этом примере метод `Main` запускает новый поток, в котором выполняется метод `Thread2`. Этот метод сохраняет значение в поле `result`, не являющемся асинхронно-изменяемым, а затем присваивает значение `true` асинхронно-изменяемому полю `finished`. Главный поток ждет, пока поле `finished` не примет значение `true`, затем читает значение поля `result`. Поскольку `finished` объявлено как `volatile`, главный поток должен прочитать из поля `result` значение 143. Если бы `finished` не было объявлено как `volatile`, сохранение значения в `result` вполне могло бы оказаться видимым в главном потоке *после* сохранения значения в `finished`, вследствие чего главный поток прочитал бы из этого поля значение `0`. Объявление поля `finished` как `volatile` предотвращает любую подобную несогласованность.

ДЖОЗЕФ АЛЬБАХАРИ

В объявлениях полей, доступ к которым всегда выполняется внутри оператора `lock` (раздел 8.12), использование ключевого слова `volatile` не требуется. Как следствие, среда выполнения должна гарантировать, что любая оптимизация упорядочивания полей, используемая между `Monitor.Enter` и `Monitor.Exit`, не выходит за пределы области видимости этих операторов.

ДЖОН СКИТ

Параллельный код, не использующий блокировки, сложен для правильного понимания. Трудно рассуждать о точных гарантиях изменчивости, в результате чего при чтении

> кода его полная корректность далеко не очевидна. Я склонен оставлять детали низкоуровневого кода, не использующего блокировки, экспертам, предпочитая вместо этого использовать написанные ими высокоуровневые библиотеки. Разумеется, важно, чтобы спецификация все-таки содержала детали поведения, которые будут полезны этим экспертам.

10.5.4. Инициализация полей

Начальным значением поля, как статического, так и поля экземпляра, является значение по умолчанию (раздел 5.2) для типа этого поля. Невозможно получить значение поля до того, как будет произведена инициализация по умолчанию, поэтому поле не может быть «неинициализированным». Пример:

```
using System;

class Test
{
   static bool b;
   int i;

   static void Main() {
      Test t = new Test();
      Console.WriteLine("b = {0}, i = {1}", b, t.i);
   }
}
```

Этот пример выведет

```
b = False, i = 0
```

так как `b` и `i` автоматически инициализируются своими значениями по умолчанию.

10.5.5. Инициализаторы переменных

Объявления полей могут содержать *инициализаторы-переменных*. Для статических полей эти инициализаторы соответствуют операторам присваивания, выполняющимся в процессе инициализации класса. Для полей экземпляра они соответствуют операторам присваивания, выполняющимся при создании экземпляра класса. Пример:

```
using System;

class Test
{
   static double x = Math.Sqrt(2.0);
   int i = 100;
   string s = "Привет";

   static void Main() {
      Test a = new Test();
      Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
   }
}
```

Этот пример выведет

```
x = 1.4142135623731, i = 100, s = Привет
```

так как присваивание значения x происходит при выполнении инициализаторов статического поля, а присваивания i и s — при выполнении инициализаторов полей экземпляра.

Инициализация значением по умолчанию, описанная в разделе 10.5.4, выполняется для всех полей, включая поля, содержащие инициализаторы переменных. Таким образом, при инициализации класса все его статические поля сначала инициализируются своими значениями по умолчанию, а затем в текстовом порядке выполняются инициализаторы статических полей. Аналогично, при создании экземпляра класса все поля этого экземпляра сначала инициализируются своими значениями по умолчанию, а затем в текстовом порядке выполняются инициализаторы полей экземпляра.

Статические поля, имеющие инициализаторы переменных, можно застать в состоянии, когда им присвоено значение по умолчанию, но использование такого кода не рекомендуется с точки зрения стиля. Пример:

```
using System;
class Test
{
   static int a = b + 1;
   static int b = a + 1;

   static void Main() {
       Console.WriteLine("a = {0}, b = {1}", a, b);
   }
}
```

Здесь продемонстрирована описанная ситуация. Несмотря на циклические определения a и b, программа является корректной. Она выводит

```
a = 1, b = 2
```

поскольку, перед тем как будут выполнены инициализаторы статических полей a и b, эти поля инициализируются значениями 0 (значение по умолчанию для int). Когда выполняется инициализатор для a, b содержит значение 0, поэтому a инициализируется значением 1. Когда выполняется инициализатор для b, a уже содержит значение 1, поэтому b инициализируется значением 2.

БИЛЛ ВАГНЕР

Этот случай проще понять, когда статические переменные объявляются в частичных классах в нескольких единицах компиляции. Более того, поскольку вы не знаете, в каком порядке будут включаться исходные единицы, вы не можете знать, какая из переменных будет проинициализирована первой.

10.5.5.1. Инициализация статических полей

Инициализаторы переменных в статических полях класса представляют собой последовательность присваиваний, выполняемых в текстовом порядке, в котором

они расположены в объявлении класса. Если в классе существует статический конструктор (раздел 10.12), выполнение инициализаторов статических полей происходит непосредственно перед выполнением этого конструктора. В противном случае точное время их выполнения зависит от реализации, но они выполняются до первого использования статического поля в данном классе. Пример:

```
using System;

class Test
{
   static void Main() {
       Console.WriteLine("{0} {1}", B.Y, A.X);
   }

   public static int F(string s) {
      Console.WriteLine(s);
      return 1;
   }
}

class A
{
    public static int X = Test.F("Инициализация A");
}

class B
{
    public static int Y = Test.F("Инициализация B");
}
```

Этот пример может вывести либо

```
Инициализация A
Инициализация B
1 1
```

либо

```
Инициализация B
Инициализация A
1 1
```

Два варианта возможны потому, что инициализаторы `X` и `Y` могут выполняться в произвольном порядке; единственное ограничение заключается в том, что выполнение должно выполняться перед обращениями к данным полям. Рассмотрим, однако, еще один пример:

```
using System;
class Test
{
   static void Main() {
       Console.WriteLine("{0} {1}", B.Y, A.X);
   }
      public static int F(string s) {
      Console.WriteLine(s);
      return 1;
   }
}
```

продолжение ☞

```
class A
{
   static A() {}
   public static int X = Test.F("Инициализация A");
}

class B
{
   static B() {}
   public static int Y = Test.F("Инициализация B");
}
```

Этот пример однозначно выведет

```
Инициализация B
Инициализация A
1 1
```

так как в соответствии с правилами, описанными в разделе 10.12, при выполнении статических конструкторов статический конструктор `B` (а следовательно, и инициализаторы статических полей `B`) должен выполняться перед статическим конструктором и инициализаторами полей `A`.

10.5.5.2. Инициализация полей экземпляра

Инициализаторы переменных в полях экземпляра класса представляют собой последовательность присваиваний, выполняемых непосредственно при входе в любой из конструкторов экземпляра (раздел 10.11.1) этого класса. Инициализаторы переменных выполняются в текстовом порядке, в котором они расположены в объявлении класса. Более подробно процесс создания экземпляра класса и инициализации описан в разделе 10.11.

Инициализатор переменной для поля экземпляра не может ссылаться на создаваемый экземпляр. Таким образом, использование `this` в инициализаторе приведет к ошибке компиляции, поскольку обращение в инициализаторе переменной к любому элементу экземпляра через *простое-имя* является ошибкой. Пример:

```
class A
{
   int x = 1;
   int y = x + 1; // Ошибка: обращение к элементу экземпляра this
}
```

Здесь инициализатор переменной для `y` приводит к ошибке компиляции, так как он ссылается на элемент создаваемого экземпляра.

10.6. Методы

Метод — это элемент, реализующий вычисление или действие, которое может выполнить объект или класс. Методы объявляются с помощью *объявлений-методов*.

```
объявление-метода:
   заголовок-метода   тело-метода
```

```
заголовок-метода:
   атрибуты_opt   модификаторы-метода_opt   partial_opt
        тип-возвращаемого-значения   имя-элемента   список-параметров-типов_opt
            (  список-формальных-параметров_opt  )
               ограничения-на-параметры-типы_opt

модификаторы-метода:
   модификатор-метода
   модификаторы-метода   модификатор-метода

модификатор-метода:
   new
   public
   protected
   internal
   private
   static
   virtual
   sealed
   override
   abstract
   extern

тип-возвращаемого-значения:
   тип
   void

имя-элемента:
   идентификатор
   интерфейс   .   идентификатор

тело-метода:
   блок
   ;
```

Объявление-метода может содержать набор *атрибутов* (раздел 17) и корректную комбинацию четырех модификаторов доступа (раздел 10.3.5) и модификаторов `new` (раздел 10.3.4), `static` (раздел 10.6.2), `virtual` (раздел 10.6.3), `override` (раздел 10.6.4), `sealed` (раздел 10.6.5), `abstract` (раздел 10.6.6) и `extern` (раздел 10.6.7).

Комбинация модификаторов в объявлении считается корректной, если выполняются все перечисленные условия:

- Объявление содержит корректную комбинацию модификаторов доступа (раздел 10.3.5).
- Один и тот же модификатор не встречается в объявлении несколько раз.
- Объявление содержит не более одного из следующих модификаторов: `static`, `virtual` и `override`.
- Объявление содержит не более одного из следующих модификаторов: `new` и `override`.
- Если объявление содержит модификатор `abstract`, то оно не содержит ни одного из следующих модификаторов: `static`, `virtual`, `sealed` и `extern`.

- Если объявление содержит модификатор `private`, то оно не содержит ни одного из следующих модификаторов: `virtual`, `override` и `abstract`.
- Если объявление содержит модификатор `sealed`, оно также содержит и модификатор `override`.
- Если объявление содержит модификатор `partial`, то оно не содержит ни одного из следующих модификаторов: `new`, `public`, `protected`, `internal`, `private`, `virtual`, `sealed`, `override`, `abstract` и `extern`.

Тип-возвращаемого-значения в объявлении метода определяет тип значения, которое вычисляется и возвращается методом. Если метод не возвращает значения, *типом-возвращаемого-значения* является `void`. Метод должен возвращать `void` и в случае, если объявление содержит модификатор `partial`.

Имя-элемента определяет имя метода. Если метод не является явной реализацией элемента интерфейса (раздел 13.4.1), *имя-элемента* — это просто *идентификатор*. *Имя-элемента* для явной реализации элемента интерфейса состоит из *интерфейса*, символа «`.`» и *идентификатора*.

Необязательный *список-параметров-типов* определяет параметры-типы для метода (раздел 10.1.3). Если этот список указан, метод является **обобщенным методом**. Если для метода указан модификатор `extern`, указание *списка-параметров-типов* недопустимо.

Необязательный *список-формальных-параметров* определяет параметры метода (раздел 10.6.1).

Необязательные *ограничения-на-параметры-типы* определяют ограничения на отдельные параметры-типы (раздел 10.1.5) и могут указываться, только если в методе также присутствует *список-параметров-типов* и метод не содержит модификатора `override`.

Тип-возвращаемого-значения и все типы, указанные в *списке-формальных-параметров* метода, должны иметь не более строгий вид доступа, чем сам метод (раздел 3.5.4).

Для методов, объявленных с модификатором `abstract` или `extern`, *тело-метода* состоит только из точки с запятой. Для методов, объявленных как `partial`, *тело-метода* может состоять или из точки с запятой, или из блока. Для всех остальных методов *тело-метода* состоит из *блока*, определяющего операторы, выполняемые при вызове метода.

Имя, список параметров-типов и список формальных параметров метода определяют его сигнатуру (раздел 3.6). Точнее, сигнатура метода состоит из его имени, количества параметров-типов, а также количества, модификаторов и типов его формальных параметров. Любой параметр-тип в методе, встречающийся в типе формального параметра, определяется не по своему имени, а по порядковому номеру в списке аргументов-типов метода. Тип возвращаемого значения не является частью сигнатуры метода, равно как и имена параметров-типов или формальных параметров.

Имя метода должно отличаться от имен всех остальных элементов, не являющихся методами и объявленных в том же классе. Кроме этого, сигнатура метода должна отличаться от сигнатур всех остальных методов, объявленных в том же

классе, а сигнатуры двух методов, объявленных в одном классе, не могут отличаться только модификаторами `ref` и `out`.

Параметры-типы метода находятся в области видимости во всем *объявлении-метода* и могут быть использованы для формирования типов для *типа-возвращаемого-значения*, *тела-метода* и *ограничений-на-параметры-типы*, но не для *атрибутов*.

Имена всех формальных параметров и параметров-типов должны отличаться.

10.6.1. Параметры метода

Параметры метода, если они есть, объявляются *списком-формальных-параметров* метода.

список-формальных-параметров:
  фиксированные-параметры
  фиксированные-параметры **,** *параметр-массив*
  параметр-массив

фиксированные-параметры:
  фиксированный-параметр
  фиксированные-параметры **,** *фиксированный-параметр*

фиксированный-параметр:
  атрибуты$_{opt}$ *модификатор-параметра*$_{opt}$ *тип* *идентификатор*
    аргумент-по-умолчанию$_{opt}$

аргумент-по-умолчанию:
  = *выражение*

модификатор-параметра:
  ref
  out
  this

параметр-массив:
  атрибуты$_{opt}$ **params** *тип-массива* *идентификатор*

Список формальных параметров состоит из одного или нескольких параметров, разделенных запятыми, при этом только последний из них может являться *параметром-массивом*.

Фиксированный-параметр состоит из необязательного набора *атрибутов* (раздел 17), необязательного модификатора `ref`, `out` или `this`, *типа*, *идентификатора* и необязательного *аргумента-по-умолчанию*. Каждый *фиксированный-параметр* объявляет параметр заданного типа с заданным именем. Модификатор `this` объявляет метод методом расширения и может использоваться только для первого параметра статического метода. Подробнее методы расширения описаны в разделе 10.6.9.

Фиксированный-параметр, имеющий *аргумент-по-умолчанию*, называется **необязательным параметром**, а *фиксированный-параметр*, не имеющий такого аргумента, — **обязательным параметром**. Обязательный параметр не может располагаться после необязательного в *списке-формальных-параметров*.

КРИСТИАН НЕЙГЕЛ

Необязательным параметрам должно быть уделено особое внимание при управлении версиями. Компилятор добавляет аргументы по умолчанию в вызывающий код в сборке, где этот код находится. Если значение аргумента по умолчанию в новой версии кода меняется, вызывающий код все еще содержит старое значение, если он не был перекомпилирован.

Параметр `ref` или `out` не может иметь *аргумента-по-умолчанию*. *Выражением* в *аргументе-по-умолчанию* должно быть одно из следующего:

- *Константное-выражение*.
- Выражение в форме `new S()`, где `S` — тип-значение.
- Выражение в форме `default(S)`, где `S` — тип-значение.

Выражение должно быть неявно преобразуемо в тип параметра с помощью тождественного или обнуляемого преобразования.

Если необязательные параметры содержатся в реализующем объявлении частичного метода (раздел 10.2.7), в явной реализации элемента интерфейса (раздел 13.4.1) или в объявлении индексатора с одним параметром (раздел 10.9), компилятор должен выдать предупреждение, так как эти элементы никогда не могут быть вызваны способом, допускающим пропуск аргументов.

Параметр-массив состоит из необязательного набора *атрибутов* (раздел 17), модификатора `params`, *типа-массива* и *идентификатора*. Параметр-массив объявляет один параметр заданного типа массива с заданным именем. *Типом-массива* для этого параметра должен быть тип одномерного массива (раздел 12.1). При вызове метода параметр-массив позволяет указывать либо один аргумент данного типа массива, либо произвольное количество (в том числе ни одного) аргументов, имеющих тип элемента массива. Более подробно параметры-массивы рассматриваются в разделе 10.6.1.4.

Параметр-массив может располагаться после необязательного параметра, но не может иметь значение по умолчанию — отсутствие аргументов для *параметра-массива* приводит к созданию пустого массива.

Следующий пример демонстрирует различные типы параметров:

```
public void M(
    ref int     i,
    decimal     d,
    bool        b = false,
    bool?       n = false,
    string      s = "Привет",
    object      o = null,
    T           t = default(T),
    params      int[] a
) { }
```

В *списке-формальных-параметров* для `M` `i` является обязательным параметром `ref`, `d` — обязательным параметром-значением, `b`, `s`, `o` и `t` — необязательными параметрами-значениями, `a` — параметром-массивом.

Объявление метода создает отдельную область объявлений для параметров, параметров-типов и локальных переменных. Имена вводятся в эту область списком

параметров-типов, списком формальных параметров и объявлениями локальных переменных в *блоке* метода. Если два элемента в области объявлений метода имеют одинаковые имена, возникает ошибка. Если область объявлений метода и область объявлений локальных переменных во вложенной области объявлений содержат элементы с одинаковыми именами, возникает ошибка.

При вызове метода (раздел 7.6.5.1) создается отдельная для данного вызова копия формальных параметров и локальных переменных метода, и только что созданным формальным параметрам присваиваются значения или ссылки на переменные из списка аргументов вызова. Внутри *блока* метода на формальные параметры можно ссылаться с помощью их идентификаторов с помощью выражений вида *простое-имя* (раздел 7.6.2).

Существуют четыре разновидности формальных параметров:

- Параметры-значения, объявляемые без каких-либо модификаторов.
- Параметры-ссылки, объявляемые с модификатором `ref`.
- Выходные параметры, объявляемые с модификатором `out`.
- Параметры-массивы, объявляемые с модификатором `params`.

Как указано в разделе 3.6, модификаторы `ref` и `out` являются частью сигнатуры метода, а модификатор `params` — нет.

10.6.1.1. Параметры-значения

Параметр, объявленный без модификаторов, является параметром-значением. Параметр-значение соответствует локальной переменной, получающей свое начальное значение из соответствующего аргумента, переданного при вызове метода.

Когда формальный параметр является параметром-значением, соответствующий аргумент в вызове метода должен быть выражением, неявно преобразуемым (раздел 6.1) к типу формального параметра.

Метод может присваивать новые значения параметру-значению. Такие присваивания оказывают влияние только на локальную область памяти, представленную этим параметром; они не влияют на фактический аргумент, указанный при вызове метода.

10.6.1.2. Параметры-ссылки

Параметр, объявленный с модификатором `ref`, является параметром-ссылкой. В отличие от параметра-значения, параметр-ссылка не создает новую область памяти. Вместо этого параметр-ссылка представляет ту же самую область памяти, что и переменная, переданная в качестве аргумента при вызове метода.

Когда формальный параметр является параметром-ссылкой, соответствующий аргумент в вызове метода должен состоять из ключевого слова `ref`, за которым следует *ссылка-на-переменную* (раздел 5.3.3) того же типа, что и формальный параметр. Переменная должна быть явно присвоена перед тем, как ее можно будет передавать в качестве параметра-ссылки.

Внутри метода параметр-ссылка всегда считается явно присвоенным.

Метод, объявленный как итератор (раздел 10.14), не может содержать параметры-ссылки.

Пример:

```
using System;

class Test
{
   static void Swap(ref int x, ref int y) {
      int temp = x;
      x = y;
      y = temp;
   }

   static void Main() {
      int i = 1, j = 2;
      Swap(ref i, ref j);
      Console.WriteLine("i = {0}, j = {1}", i, j);
   }
}
```

Этот пример выведет следующее:

```
i = 2, j = 1
```

При вызове `Swap` в `Main` `x` представляет `i`, а `y` представляет `j`. Таким образом, результатом вызова является обмен значениями между `i` и `j`.

В методе, принимающем параметры-ссылки, допустимо указывать несколько имен, представляющих одну и ту же область памяти. Пример:

```
class A
{
   string s;

   void F(ref string a, ref string b) {
      s = "Один";
      a = "Два";
      b = "Три";
   }

   void G() {
       F(ref s, ref s);
   }
}
```

Здесь вызов `F` в `G` передает ссылку на `s` в виде как `a`, так и `b`. Таким образом, для этого вызова имена `s`, `a` и `b` ссылаются на одну и ту же область памяти, и все три присваивания изменяют поле экземпляра `s`.

10.6.1.3. Выходные параметры

Параметр, объявленный с модификатором `out`, является выходным параметром. Так же как и параметр-ссылка, выходной параметр не создает новую область памяти. Выходной параметр представляет ту же самую область памяти, что и переменная, переданная в качестве аргумента при вызове метода.

Когда формальный параметр является выходным параметром, соответствующий аргумент в вызове метода должен состоять из ключевого слова `out`, за которым следует *ссылка-на-переменную* (раздел 5.3.3) того же типа, что и формальный параметр. Не требуется, чтобы переменная была явно присвоена перед тем, как она будет передана в качестве выходного параметра, однако после вызова, в который она была передана таким способом, данная переменная считается явно присвоенной.

Внутри метода выходной параметр аналогично локальной переменной первоначально считается неинициализированным, и перед использованием его значения он должен быть явно присвоен.

Каждый выходной параметр в методе должен быть явно присвоен перед возвратом из метода.

Метод, объявленный как частичный метод (раздел 10.2.7) или итератор (раздел 10.14), не может содержать выходные параметры.

Выходные параметры обычно используются в методах, возвращающих несколько значений.

БИЛЛ ВАГНЕР

Разумеется, для возврата нескольких значений вы могли бы создать `struct` или `class`, избежав тем самым необходимости использовать выходные параметры. Кроме этого, для возврата нескольких значений можно использовать новые обобщенные классы `Tuple<>`.

ДЖОН СКИТ

В качестве примера к комментарию Билла можно привести метод `int.TryParse` платформы .NET. Фактически он возвращает два значения: проанализированное целочисленное значение и булевский флаг, обозначающий успешность операции. Обнуляемые типы-значения в C# предоставляют точно такую же возможность, поэтому текущая сигнатура метода:

```
bool TryParse(string s, out int result)
```

могла бы быть заменена на:

```
int? TryParse(string s)
```

То же самое верно и для аналогичных вызовов, например `decimal.TryParse`.

Пример:

```
using System;

class Test
{
   static void SplitPath(string path, out string dir, out string name) {
      int i = path.Length;
      while (i > 0) {
         char ch = path[i – 1];
         if (ch == '\\' || ch == '/' || ch == ':') break;
         i--;
      }
```

продолжение ☞

```
      dir = path.Substring(0, i);
      name = path.Substring(i);
   }

   static void Main() {
      string dir, name;
      SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
      Console.WriteLine(dir);
      Console.WriteLine(name);
   }
}
```

Этот пример выведет следующее:

```
c:\Windows\System\
hello.txt
```

Заметьте, что переменные `dir` и `name` могут быть неинициализированными к моменту их передачи в метод `SplitPath` и что в коде, следующем за вызовом, они считаются инициализированными.

10.6.1.4. Параметры-массивы

Параметр, объявленный с модификатором `params`, является параметром-массивом. Если список формальных параметров содержит параметр-массив, этот параметр должен быть последним в списке и иметь тип одномерного массива. Например, типы `string[]` и `string[][]` могут использоваться в качестве типа параметра-массива, а тип `string[,]` — не может. Невозможно использовать модификатор `params` совместно с модификаторами `ref` и `out`.

При вызове метода параметр-массив допускает передачу аргументов одним из двух способов:

- Аргумент, переданный в качестве параметра-массива, может представлять собой одиночное выражение, неявно преобразуемое (раздел 6.1) в тип параметра-массива. В этом случае параметр-массив действует в точности так же, как параметр-значение.
- Альтернативным вариантом является указание для параметра-массива произвольного количества (в том числе и ни одного) аргументов, каждый из которых представлен выражением, неявно преобразуемым (раздел 6.1) в тип элемента параметра-массива. В этом случае вызов создает экземпляр типа параметра-массива с размером, соответствующим числу аргументов, инициализирует элементы экземпляра массива значениями переданных аргументов и использует только что созданный экземпляр массива в качестве фактического аргумента.

За исключением возможности указывать при вызове переменное количество аргументов, параметр-массив в точности эквивалентен параметру-значению (раздел 10.6.1.1) того же типа.

Пример:

```
using System;

class Test
{
```

```
   static void F(params int[] args) {
      Console.Write("Массив содержит {0} элементов:", args.Length);
      foreach (int i in args)
              Console.Write(" {0}", i);
      Console.WriteLine();
   }

   static void Main() {
      int[] arr = {1, 2, 3};
      F(arr);
      F(10, 20, 30, 40);
      F();
   }
}
```

Этот пример выведет следующее:

```
Массив содержит 3 элементов: 1 2 3
Массив содержит 4 элементов: 10 20 30 40
Массив содержит 0 элементов:
```

В первом вызове `F` ему просто передается массив `a` в качестве параметра-значения. Во втором вызове `F` автоматически создается экземпляр массива типа `int[]` из четырех элементов с указанными значениями и передается в качестве параметра-значения. Аналогично, в третьем вызове `F` автоматически создается экземпляр пустого массив `int[]`, который передается в качестве параметра-значения. Второй и третий вызовы в точности эквивалентны следующим:

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

При осуществлении разрешения перегрузки метод с параметром-массивом может подходить либо в своей нормальной форме, либо в расширенной форме (раздел 7.5.3.1). Расширенная форма метода используется, только если нормальная форма не подошла и если в том же типе не объявлен метод, имеющий ту же сигнатуру, что и расширенная форма.

Пример:

```
using System;

class Test
{
   static void F(params object[] a) {
       Console.WriteLine("F(object[])");
   }

   static void F() {
       Console.WriteLine("F()");
   }

   static void F(object a0, object a1) {
       Console.WriteLine("F(object,object)");
   }

   static void Main() {
      F();
      F(1);
```

продолжение ⇗

```
      F(1, 2);
      F(1, 2, 3);
      F(1, 2, 3, 4);
   }
}
```

Этот пример выведет следующее:

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

> **БИЛЛ ВАГНЕР**
>
> Чем больше потенциально подходящих методов вы создаете, тем больше вы создаете проблем пользователям вашего кода. Бóльшая неоднозначность для компилятора означает также и бóльшую неоднозначность для разработчиков.

В приведенном примере две из возможных расширенных форм метода с параметром-массивом уже присутствуют в классе в виде обычных методов. Таким образом, эти расширенные формы не рассматриваются при разрешении перегрузки, и в первом и третьем вызовах выбираются обычные методы. Когда в классе объявлен метод с параметром-массивом, нередким является наличие в нем также некоторых расширенных форм метода в виде обычных методов. Такой подход позволяет избежать выделения экземпляра массива, происходящего при вызове расширенной формы метода с параметром-массивом.

Когда параметр-массив имеет тип `object[]`, возникает потенциальная двусмысленность при выборе между нормальной и расширенной формами метода в случае передачи ему одного параметра типа `object`. Причиной двусмысленности является тот факт, что `object[]` сам по себе неявно преобразуем в тип `object`. Она, однако, не представляет проблемы, так как может быть в случае необходимости разрешена путем добавления приведения типа.

Пример:

```
using System;

class Test
{
   static void F(params object[] args) {
      foreach (object o in args) {
         Console.Write(o.GetType().FullName);
         Console.Write(" ");
      }
      Console.WriteLine();
   }

   static void Main() {
      object[] a = {1, "Привет", 123.456};
      object o = a;
```

```
        F(a);
    F((object)a);
    F(o);
    F((object[])o);
  }
}
```

Этот пример выведет следующее:

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

При первом и последнем вызовах `F` подходит его нормальная форма, так как существует неявное преобразование из типа аргумента в тип параметра (оба имеют тип `object[]`). Следовательно, при разрешении перегрузки выбирается нормальная форма `F`, и аргумент передается как обычный параметр-значение. При втором и третьем вызовах нормальная форма не применима, так как не существует неявного преобразования из типа аргумента в тип параметра (тип `object` невозможно неявно преобразовать в тип `object[]`). Однако применима расширенная форма `F`, поэтому при разрешении перегрузки выбирается именно она. В результате при вызове создается массив `object[]` с одним элементом, и этот единственный элемент инициализируется указанным значением аргумента (которое представляет собой ссылку на `object[]`).

10.6.2. Статические методы и методы экземпляра

Когда объявление метода содержит модификатор `static`, метод называется статическим методом. Если модификатор `static` отсутствует, метод называется методом экземпляра.

Статический метод не работает с конкретным экземпляром, поэтому попытка сослаться на `this` в таком методе приведет к ошибке компиляции.

Метод экземпляра работает с заданным экземпляром класса и может получать доступ к этому экземпляру с помощью `this` (раздел 7.6.7).

Когда к методу обращаются с помощью *доступа-к-элементу* (раздел 7.6.4) в форме `E.M` и `M` — статический метод, `E` должен обозначать тип, содержащий `M`; если `M` — метод экземпляра, `E` должен обозначать экземпляр типа, содержащего `M`.

Более подробно различия между статическими элементами и элементами экземпляра рассматриваются в разделе 10.3.7.

10.6.3. Виртуальные методы

Когда объявление метода экземпляра содержит модификатор `virtual`, этот метод называется виртуальным методом. Если модификатор `virtual` отсутствует, метод называется невиртуальным.

БИЛЛ ВАГНЕР

Вы можете создавать виртуальные обобщенные методы даже в классе, не являющемся обобщенным. Когда вы это делаете, любые переопределения метода также должны быть обобщенными. Вы не можете переопределить конкретную реализацию.

ДЖОН СКИТ

Закрытые методы не могут быть виртуальными, хотя и существует один редкий случай, в котором это имело бы смысл: такой метод мог бы быть переопределен (другим закрытым методом) во вложенном классе, унаследованном от содержащего его класса. Хотя запрещать такую возможность в спецификации немного неэлегантно, этот запрет означает, что виртуальные методы, *случайно* объявленные закрытыми, могут считаться ошибкой, а это гораздо более частый случай.

Реализация невиртуального метода инвариантна: она является той же самой вне зависимости от того, вызывался ли метод для экземпляра класса, в котором он был объявлен, или же для экземпляра производного класса. В противоположность этому, реализация виртуального метода может быть замещена в производных классах. Процесс замещения реализации унаследованного виртуального метода называется **переопределением** этого метода (раздел 10.6.4).

При вызове виртуального метода его фактическая реализация, которая будет вызвана, определяется **типом времени выполнения** экземпляра, для которого выполняется вызов. При вызове невиртуального метода таким определяющим фактором является **тип времени компиляции**. Точнее говоря, когда метод с именем `N` вызывается со списком аргументов `A` для экземпляра, имеющего тип времени компиляции `C` и тип времени выполнения `R` (где `R` — либо `C`, либо класс, унаследованный от `C`), вызов выполняется следующим образом:

- Сначала к `C`, `N` и `A` применяется разрешение перегрузки, чтобы выбрать конкретный метод `M` из множества методов, объявленных в `C`, а также унаследованных им. Этот процесс описан в разделе 7.6.5.1.
- Затем, если `M` — невиртуальный метод, вызывается `M`.
- В противном случае `M` — виртуальный метод, и вызывается реализация самого нижнего уровня относительно `R`.

Для каждого виртуального метода, объявленного в классе или унаследованного им, существует его **реализация самого нижнего уровня** относительно данного класса. Реализация виртуального метода `M` самого нижнего уровня относительно класса `R` определяется следующим образом:

- Если `R` содержит первоначальное объявление метода `M` с модификатором `virtual`, оно является реализацией `M` самого нижнего уровня.
- В противном случае, если `R` содержит переопределение `M` (с модификатором `override`), оно является реализацией `M` самого нижнего уровня.
- В противном случае реализацией `M` самого нижнего уровня относительно `R` является реализация `M` самого нижнего уровня относительно непосредственного базового класса `R`.

Следующий пример демонстрирует различия между виртуальными и невиртуальными методами:

```
using System;

class A
{
   public void F() { Console.WriteLine("A.F"); }
   public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A
{
   new public void F() { Console.WriteLine("B.F"); }
   public override void G() { Console.WriteLine("B.G"); }
}

class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      a.F();
      b.F();
      a.G();
      b.G();
   }
}
```

В этом примере класс `A` вводит невиртуальный метод `F` и виртуальный метод `G`. Класс `B` вводит *новый* невиртуальный метод `F`, таким образом *скрывая* унаследованный `F`, а также *переопределяет* унаследованный метод `G`. Результатом будет следующий вывод:

```
A.F
B.F
B.G
B.G
```

Заметьте, что оператор `a.G()` вызывает `B.G`, а не `A.G`. Это происходит вследствие того, что фактическая реализация, которая будет вызвана, определяется типом времени выполнения экземпляра (в данном случае это `B`), а не типом времени компиляции (в данном случае `A`).

Так как методы могут скрывать унаследованные методы, класс может содержать несколько виртуальных методов с одинаковыми сигнатурами. Двусмысленность в данном случае не возникает, поскольку все методы, кроме метода самого нижнего уровня, скрыты. Пример:

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}
```

продолжение ⇨

```
class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
   static void Main() {
      D d = new D();
      A a = d;
      B b = d;
      C c = d;
      a.F();
      b.F();
      c.F();
      d.F();
   }
}
```

В этом примере классы `C` и `D` содержат по два виртуальных метода с одинаковыми сигнатурами: первый введен в `A`, а второй — в `C`. Метод, введенный в `C`, скрывает метод, унаследованный от `A`. Таким образом, объявление `override` в `D` переопределяет метод, введенный в `C`, при этом в `D` невозможно переопределить метод, введенный в `A`. Пример выведет следующее:

```
B.F
B.F
D.F
D.F
```

Заметьте, что скрытый виртуальный метод можно вызвать, обратившись к экземпляру `D` через тип более высокого уровня, в котором этот метод не скрыт.

10.6.4. Переопределенные методы

Когда объявление метода экземпляра содержит модификатор `override`, метод называется **переопределенным методом**. Переопределенный метод переопределяет унаследованный виртуальный метод с такой же сигнатурой. Тогда как объявление виртуального метода *вводит* новый метод, объявление переопределенного метода *уточняет* существующий унаследованный виртуальный метод, предоставляя его новую реализацию.

Метод, переопределяемый с помощью объявления `override`, называется **переопределяемым базовым методом**. Для переопределенного метода `M`, объявленного в классе `C`, переопределяемый базовый метод определяется путем анализа каждого

базового класса C, начиная с непосредственного базового класса C и далее проверяя каждый последующий непосредственный базовый класс, пока в некотором базовом классе не будет найден хотя бы один доступный метод, имеющий такую же, как и M, сигнатуру после подстановки аргументов типа. В процессе поиска метод считается доступным, если он объявлен с модификатором public, protected или protected internal или если он объявлен с модификатором internal в той же программе, что и C.

Если для переопределяющего объявления не выполняется хотя бы одно из перечисленных ниже условий, возникает ошибка компиляции:

- Переопределяемый базовый метод может быть найден описанным выше способом.
- Существует ровно один такой переопределяемый базовый метод. Данное ограничение применимо, только если тип базового класса является сконструированным типом, где подстановка аргументов-типов делает сигнатуры двух методов одинаковыми.
- Переопределяемый базовый метод является виртуальным, абстрактным или переопределенным. Иными словами, такой метод не может быть статическим или невиртуальным.
- Переопределяемый базовый метод не является бесплодным.
- Переопределенный метод и переопределяемый базовый метод имеют один и тот же тип возвращаемого значения.
- Переопределяющее объявление и переопределяемый базовый метод имеют один и тот же вид доступа. Иными словами, переопределяющее объявление не может изменить вид доступа виртуального метода. Однако если переопределяемый базовый метод объявлен как protected internal и расположен в сборке, отличной от той, в которой содержится переопределенный метод, последний должен быть объявлен как protected.
- Переопределяющее объявление не содержит ограничений-на-параметры-типы. Напротив, ограничения наследуются от переопределяемого базового метода. Заметьте, что ограничения, представляющие собой параметры-типы в переопределяемом методе, могут быть заменены аргументами-типами в унаследованном ограничении. Это может привести к появлению ограничений, являющихся некорректными при их явном указании, таких как типы-значения или бесплодные типы.

Следующий пример демонстрирует действие правил переопределения для обобщенных классов:

```
abstract class C<T>
{
   public virtual T F() {...}

   public virtual C<T> G() {...}

   public virtual void H(C<T> x) {...}
}
```

продолжение ⇗

```
class D: C<string>
{
   public override string F() {...}          // Все в порядке

   public override C<string> G() {...}       // Все в порядке

   public override void H(C<T> x) {...}      // Ошибка: требуется C<string>
}

class E<T,U>: C<U>
{
   public override U F() {...}               // Все в порядке

   public override C<U> G() {...}            // Все в порядке

   public override void H(C<T> x) {...}      // Ошибка: требуется C<U>
}
```

Из переопределяющего объявления можно получить доступ к переопределяемому базовому методу с помощью *base-доступа* (раздел 7.6.8). Пример:

```
class A
{
   int x;

   public virtual void PrintFields() {
       Console.WriteLine("x = {0}", x);
   }
}

class B: A
{
   int y;

   public override void PrintFields() {
      base.PrintFields();
      Console.WriteLine("y = {0}", y);
   }
}
```

Вызов `base.PrintFields()` в `B` вызывает метод `PrintFields`, объявленный в `A`. *Base-доступ* отключает механизм виртуальных вызовов и просто рассматривает базовый метод как невиртуальный. Если бы вызов в `B` был записан как `((A)this).PrintFields()`, он бы рекурсивно вызывал метод `PrintFields`, объявленный в `B`, а не в `A`, поскольку `PrintFields` является виртуальным, а тип времени выполнения `((A)this)` — это `B`.

Метод может переопределить другой метод только с помощью модификатора `override`. Во всех остальных случаях метод с той же сигнатурой, что и унаследованный, просто скрывает последний.

Пример:

```
class A
{
    public virtual void F() {}
}
```

```
class B: A
{
    public virtual void F() {}       // Предупреждение: скрытие
                                     // унаследованного метода F()
}
```

Здесь метод F в B не содержит модификатора override и, следовательно, не переопределяет метод F в A. Вместо этого он скрывает метод, объявленный в A, а поскольку объявление не содержит модификатора new, компилятор выдает предупреждение.

Пример:

```
class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {}            // Скрывает A.F в пределах тела B
}

class C: B
{
    public override void F() {}        // Все в порядке: переопределяет A.F
}
```

Здесь метод F в B скрывает виртуальный метод F, унаследованный от A. Так как новый метод F в B имеет вид доступа private, его область видимости распространяется только на тело класса B и не распространяется на C. Следовательно, объявление F в C может переопределять метод F, унаследованный от A.

10.6.5. Бесплодные методы

Когда объявление метода экземпляра содержит модификатор sealed, этот метод называется **бесплодным методом**. Если объявление метода экземпляра содержит модификатор sealed, оно также должно содержать модификатор override. Использование модификатора sealed лишает производный класс возможности повторно переопределить метод.

Пример:

```
using System;
class A
{
   public virtual void F() {
       Console.WriteLine("A.F");
   }

   public virtual void G() {
       Console.WriteLine("A.G");
   }
}
```

продолжение ☞

```
class B: A
{
   sealed override public void F() {
       Console.WriteLine("B.F");
   }

   override public void G() {
       Console.WriteLine("B.G");
   }
}

class C: B
{
   override public void G() {
       Console.WriteLine("C.G");
   }
}
```

Здесь класс `B` предоставляет два переопределенных метода: метод `F` с модификатором `sealed` и метод `G` без него. Использование модификатора `sealed` в классе `B` лишает класс `C` возможности переопределить метод `F` повторно.

КРИС СЕЛЛЗ

Когда я хочу наследовать от некоторого класса, меня раздражает тот факт, что класс или метод может быть бесплодным, поскольку это ограничивает мои возможности по использованию класса.

Когда я создаю базовый класс, я хочу максимально ограничить его, чтобы иметь возможность протестировать все возможные сценарии наследования, которые я буду поддерживать. Я не хочу попасть в ловушку, когда мне придется поддерживать какой-нибудь сумасшедший код, который создал заказчик и возможность создания которого я никогда не предполагал.

Как и во всех других случаях при проектировании программного обеспечения, использование модификатора `sealed` позволяет сохранять баланс. Я предпочитаю избегать использования этого ключевого слова до тех пор, пока мне не понадобится его использовать. Если заказчики сходят с ума... Что ж, так даже интереснее.

10.6.6. Абстрактные методы

Когда объявление метода экземпляра содержит модификатор `abstract`, этот метод называется **абстрактным методом**. Хотя абстрактный метод неявно также является виртуальным, он не может содержать модификатор `virtual`.

Объявление абстрактного метода вводит новый виртуальный метод, но не предоставляет реализацию данного метода. Вместо этого неабстрактные производные классы обязаны предоставить свои собственные реализации, переопределив метод. Поскольку абстрактный метод не предоставляет фактической реализации, *тело-метода* в нем состоит лишь из точки с запятой.

Объявления абстрактных методов допустимы только в абстрактных классах (раздел 10.1.1.1).

Пример:

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}

public class Ellipse: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
       g.DrawEllipse(r);
   }
}

public class Box: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
       g.DrawRect(r);
   }
}
```

Здесь класс `Shape` определяет абстрактное представление объекта «геометрическая фигура», который может рисовать сам себя. Метод `Paint` в этом классе объявлен абстрактным, поскольку для него не существует подходящей реализации по умолчанию. Классы `Ellipse` и `Box` представляют собой конкретные реализации `Shape`. Так как эти классы не абстрактные, они обязаны переопределить метод `Paint` и предоставить фактические реализации.

Если *base-доступ* (раздел 7.6.8) ссылается на абстрактный метод, возникает ошибка компиляции. Пример:

```
abstract class A
{
    public abstract void F();
}

class B: A
{
   public override void F() {
       base.F();                // Ошибка: base.F — абстрактный
   }
}
```

Данный пример приводит к ошибке компиляции для вызова `base.F()`, поскольку в этом вызове предпринята попытка сослаться на абстрактный метод.

Объявление абстрактного метода может переопределять виртуальный метод. Это позволяет абстрактному классу заставить производные классы повторно реализовать данный метод и делает первоначальную реализацию недоступной. Пример:

```
using System;

class A
{
   public virtual void F() {
   Console.WriteLine("A.F");
   }
}
```

продолжение ⇗

```
abstract class B: A
{
   public abstract override void F();
}

class C: B
{
   public override void F() {
       Console.WriteLine("C.F");
   }
}
```

Здесь класс `A` объявляет виртуальный метод, класс `B` переопределяет этот метод абстрактным методом, а класс `C` переопределяет абстрактный метод, чтобы предоставить собственную реализацию.

МАРЕК САФАР

Не существует ограничений на видимость абстрактных методов, за исключением того, что они не могут объявляться как `private`. Отсутствие ограничений может создавать проблемы, если кто-то случайно объявит метод как `internal abstract` внутри класса, объявленного как `public abstract`; такой класс невозможно будет использовать за пределами его сборки, даже несмотря на модификатор `public`.

10.6.7. Внешние методы

Когда объявление метода содержит модификатор `extern`, этот метод называется **внешним методом**. Внешние методы реализуются за пределами программы, обычно с помощью отличного от C# языка. Поскольку объявление внешнего метода не предоставляет фактической реализации, *тело-метода* в нем состоит лишь из точки с запятой. Внешний метод не может быть обобщенным.

Модификатор `extern` обычно используется в связке с атрибутом `DllImport` (раздел 17.5.1), допуская реализацию внешних методов в библиотеках динамической компоновки (Dynamic Link Library, DLL). Среда выполнения может поддерживать и другие механизмы, с помощью которых можно предоставить реализации внешних методов.

Когда для внешнего метода используется атрибут `DllImport`, объявление метода также должно содержать модификатор `static`. Следующий пример демонстрирует использование модификатора `extern` и атрибута `DllImport`:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
   [DllImport("kernel32", SetLastError=true)]
   static extern bool CreateDirectory(string name, SecurityAttribute sa);

   [DllImport("kernel32", SetLastError=true)]
   static extern bool RemoveDirectory(string name);
```

```
    [DllImport("kernel32", SetLastError=true)]
   static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

   [DllImport("kernel32", SetLastError=true)]
   static extern bool SetCurrentDirectory(string name);
}
```

10.6.8. Частичные методы

Когда объявление метода содержит модификатор `partial`, этот метод называется **частичным методом**. Частичные методы могут быть объявлены только как элементы частичных типов (раздел 10.2), и на них накладывается ряд ограничений. Более подробно такие методы рассматриваются в разделе 10.2.7.

10.6.9. Методы расширения

Когда первый параметр метода содержит модификатор `this`, этот метод называется **методом расширения**. Методы расширения могут объявляться только в необобщенных невложенных статических классах. Первый параметр метода расширения не может содержать каких-либо модификаторов, кроме `this`, а тип параметра не может быть типом указателя.

> **ПИТЕР СЕСТОФТ**
>
> Тип параметра также не может содержать модификатор `dynamic` (хотя на самом деле он означал бы просто `object`). Но этим типом может быть `T`, где `T` — параметр-тип метода расширения, например:
>
> `public static void Foo<T>(this T x) { ... }`
>
> Это потенциально может оказаться полезным, особенно если `T` имеет ограничение типа и существуют несколько параметров, в типах которых используется `T`.

Ниже приведен пример статического класса, объявляющего два метода расширения:

```
using System;
public static class Extensions
{
   public static int ToInt32(this string s) {
       return Int32.Parse(s);
   }

   public static T[] Slice<T>(this T[] source, int index, int count) {
      if (index < 0 || count < 0 || source.Length - index < count)
              throw new ArgumentException();
      T[] result = new T[count];
      Array.Copy(source, index, result, 0, count);
      return result;
   }
}
```

Метод расширения является обычным статическим методом. Кроме того, когда охватывающий его статический класс находится в области видимости, метод расширения может быть вызван с помощью синтаксиса вызова метода экземпляра (раздел 7.6.5.2), используя в качестве первого аргумента выражение получателя.

Следующая программа использует объявленные выше методы расширения:

```
static class Program
{
   static void Main() {
      string[] strings = { "1", "22", "333", "4444" };
      foreach (string s in strings.Slice(1, 2)) {
              Console.WriteLine(s.ToInt32());
      }
   }
}
```

Метод `Slice` доступен для `string[]`, а метод `ToInt32` — для `string`, поскольку они были объявлены как методы расширения. Эта программа делает то же самое, что и приведенная ниже программа, использующая обычные вызовы статических методов:

```
static class Program
{
   static void Main() {
      string[] strings = { "1", "22", "333", "4444" };
      foreach (string s in Extensions.Slice(strings, 1, 2)) {
              Console.WriteLine(Extensions.ToInt32(s));
      }
   }
}
```

ПИТЕР СЕСТОФТ

Метод расширения `Slice` является обобщенным: он работает для любого массива с элементами типа `T` вне зависимости от типа параметра `T`. Вы также можете определять обобщенные методы расширения, использующие ограничения типов (приведенный ниже метод `IsSorted`), и методы расширения, работающие с конкретными экземплярами обобщенного типа (метод `ConcatWith`). Кроме того, в последнем методе показано, что метод расширения может, разумеется, использовать аргументы по умолчанию (раздел 10.6.1):

```
public static bool IsSorted<T>(this IEnumerable<T> xs)
    where T : IComparable<T>
{ ... }

public static string ConcatWith(this IEnumerable<String> xs,
                                string glue = ", ")
{ ... }
```

КРИС СЕЛЛЗ

С одной стороны, методы расширения представляют собой превосходный способ добавлять методы, о которых забыл проектировщик типа, сохраняя, однако, такой же

синтаксис, как если бы проектировщик учел все ваши пожелания. С другой стороны, этими методами легко злоупотребить. Однажды в экспериментальной системе я увидел метод расширения для `object`, и в нашей команде теперь постоянно шутят, смогли бы мы придумать что-нибудь, еще более сбивающее с толку, или нет.

ПИТЕР СЕСТОФТ

Вот мое предположение о наиболее широко применимом методе расширения. Любая попытка вызвать несуществующий метод `toString`, со строчной буквой `t` для любого объекта с любым количеством аргументов любого типа, вызовет вместо него этот метод:

```
public static string toString(this object x, params object[] args) {
        return "Поменяй это на ToString(), приятель!";
}
```

Методы расширения не являются виртуальными и ничего не переопределяют; поэтому невозможно заменить метод `ToString` (с прописной `T`) в существующем типе `YourType`, объявив метод расширения. Следующее объявление допустимо, но не имеет никакого эффекта. Вызов `o.ToString()` будет использовать `Object.ToString()` или некоторый метод, переопределяющий или скрывающий его, а не этот:

```
public static string ToString(this YourType x) { ... }
```

10.6.10. Тело метода

Объявление *тела-метода* состоит или из *блока*, или из точки с запятой.

Объявления абстрактных и внешних методов не предоставляют реализаций, следовательно, тела таких методов состоят просто из точки с запятой. Для всех остальных методов тело метода является блоком (раздел 8.2), содержащим операторы, выполняющиеся при вызове метода.

Когда типом возвращаемого методом значения является `void`, в операторах `return` (раздел 8.9.4) в теле этого метода не допускается указывать выражения. Если выполнение тела метода, возвращающего `void`, завершается нормально (то есть поток управления доходит до конца тела метода), управление просто возвращается в вызывающий код.

Когда тип возвращаемого методом значения отличен от `void`, в каждом операторе `return` в теле метода требуется указывать выражение, неявно преобразуемое в тип возвращаемого значения. Конечная точка тела метода, возвращающего значение, должна быть недостижима. Иными словами, в методе, возвращающем значение, поток управления не может достигать конца тела метода. Пример:

```
class A
{
   public int F() {}            // Ошибка: требуется возвращаемое значение

   public int G() {
       return 1;
   }
```

продолжение ☞

```
      public int H(bool b) {
      if (b) {
              return 1;
      }
      else {
              return 0;
      }
   }
}
```

Здесь возвращающий значение метод `F` приводит к ошибке компиляции, поскольку поток управления достигает конца тела метода. Методы `G` и `H` не содержат ошибок, поскольку все возможные пути выполнения заканчиваются операторами `return`, указывающими возвращаемые значения.

10.6.11. Перегрузка методов

Правила разрешения перегрузки для методов описаны в разделе 7.5.2.

10.7. Свойства

Свойство — это элемент, предоставляющий доступ к характеристике объекта или класса. В качестве примеров свойств можно привести длину строки, размер шрифта, заголовок окна, имя клиента и т. д. Свойства — это естественные расширения полей: и те и другие представляют собой именованные элементы, имеющие связанные с ними типы, а синтаксис для доступа к полям и свойствам идентичен. Однако в отличие от полей свойства не задают областей памяти для хранения данных, а содержат **коды доступа**, определяющие операторы, выполняемые при чтении или записи значений свойств. Таким образом, свойства предоставляют механизм для связи некоторых действий с чтением и записью атрибутов объекта; более того, они допускают вычисление этих атрибутов.

Свойства объявляются с помощью *объявлений-свойств*:

```
объявление-свойства:
   атрибуты_opt   модификаторы-свойства_opt   тип   имя-элемента
       {  объявления-кодов-доступа  }

модификаторы-свойства:
   модификатор-свойства
   модификаторы-свойства   модификатор-свойства

модификатор-свойства:
   new
   public
   protected
   internal
   private
   static
   virtual
```

```
   sealed
   override
   abstract
   extern
```

имя-элемента:
  идентификатор
  интерфейс `.` *идентификатор*

Объявление-свойства может содержать набор *атрибутов* (раздел 17) и корректную комбинацию четырех модификаторов доступа (раздел 10.3.5) и модификаторов `new` (раздел 10.3.4), `static` (раздел 10.6.2), `virtual` (раздел 10.6.3), `override` (раздел 10.6.4), `sealed` (раздел 10.6.5), `abstract` (раздел 10.6.6) и `extern` (раздел 10.6.7).

В том, что касается корректных комбинаций модификаторов, объявления свойств подчиняются тем же правилам, что и объявления методов (раздел 10.6).

Тип в объявлении свойства определяет тип свойства, вводимого объявлением, а *имя-элемента* определяет имя свойства. Если свойство не является явной реализацией элемента интерфейса, *имя-элемента* представляет собой обычный *идентификатор*. Для явной реализации элемента интерфейса (раздел 13.4.1) *имя-элемента* состоит из *интерфейса*, символа «`.`» и *идентификатора*.

Тип свойства должен иметь не более строгий вид доступа, чем само свойство (раздел 3.5.4).

Объявления-кодов-доступа, которые должны находиться между лексемами «`{`» и «`}`», объявляют коды доступа (раздел 10.7.2) свойств. Коды доступа определяют выполняемые операторы, связанные с чтением и записью свойства.

Несмотря на то что синтаксис доступа к свойству эквивалентен синтаксису доступа к полю, свойство не классифицируется как переменная. Следовательно, невозможно передать его в качестве аргумента `ref` или `out`.

Когда объявление свойства содержит модификатор `extern`, это свойство называется **внешним свойством**. Так как объявление внешнего свойства не предоставляет фактической реализации, каждое из его *объявлений-кодов-доступа* состоит из точки с запятой.

10.7.1. Статические свойства и свойства экземпляра

Когда объявление свойства содержит модификатор `static`, свойство называется **статическим свойством**. Если модификатор `static` отсутствует, оно называется **свойством экземпляра**.

Статическое свойство не связано с конкретным экземпляром, и попытка обратиться к `this` внутри кодов доступа статического свойства приведет к ошибке компиляции.

Свойство экземпляра связано с конкретным экземпляром класса, и внутри кодов доступа свойства на этот экземпляр можно ссылаться с помощью `this` (раздел 7.6.7).

Когда к свойству обращаются с помощью *доступа-к-элементу* (раздел 7.6.4) в форме `E.M` и `M` — статическое свойство, `E` должен обозначать тип, содержащий `M`; если `M` — свойство экземпляра, `E` должен обозначать экземпляр типа, содержащего `M`.

Более подробно различия между статическими элементами и элементами экземпляра рассматриваются в разделе 10.3.7.

10.7.2. Коды доступа

Объявления-кодов-доступа свойства определяют выполняемые операторы, связанные с чтением и записью этого свойства.

```
объявления-кодов-доступа:
   объявление-кода-получения   объявление-кода-установки opt
    объявление-кода-установки   объявление-кода-получения opt

объявление-кода-получения:
   атрибуты opt   модификатор-кода-доступа opt   get   тело-кода-доступа

объявление-кода-установки:
   атрибуты opt   модификатор-кода-доступа opt   set   тело-кода-доступа

модификатор-кода-доступа:
   protected
   internal
   private
   protected internal
   internal protected

тело-кода-доступа:
   блок
   ;
```

Объявления кодов доступа состоят из *объявления-кода-получения*, *объявления-кода-установки* или обоих. Каждое объявление кода доступа состоит из лексемы `get` или `set`, за которой следуют необязательный *модификатор-кода-доступа* и *тело-кода-доступа*.

На использование *модификаторов-кода-доступа* накладываются следующие ограничения:

- *Модификатор-кода-доступа* нельзя использовать в интерфейсе или явной реализации элемента интерфейса.
- Использование *модификатора-кода-доступа* для свойства или индексатора, не содержащего модификатора `override`, допускается только в том случае, если свойство или индексатор содержат оба кода доступа (`get` и `set`), и только для одного из них.
- *Модификатор-кода-доступа* для свойства или индексатора, содержащего модификатор `override`, должен совпадать с *модификатором-кода-доступа* (если он есть) для переопределяемого кода доступа.
- *Модификатор-кода-доступа* должен объявлять более строгий вид доступа, чем вид доступа, объявленный для самого свойства или индексатора. Точнее:

- Если для свойства или индексатора объявлен вид доступа `public`, *модификатор-кода-доступа* может объявляться как `protected internal`, `internal`, `protected` или `private`.
- Если для свойства или индексатора объявлен вид доступа `protected internal`, *модификатор-кода-доступа* может объявляться как `internal`, `protected` или `private`.
- Если для свойства или индексатора объявлен вид доступа `internal` или `protected`, *модификатор-кода-доступа* может объявляться как `private`.
- Если для свойства или индексатора объявлен вид доступа `private`, использование *модификатора-кода-доступа* недопустимо.

Для абстрактных и внешних свойств *тело-кода-доступа* для каждого указанного кода доступа представляет собой просто точку с запятой. Свойство, не являющееся абстрактным или внешним, может быть **автоматически реализуемым свойством**, в этом случае должны быть представлены оба кода доступа, а их тела должны представлять собой точки с запятой (раздел 10.7.3). Для кодов доступа всех остальных свойств, не являющихся абстрактными или внешними, *тело-кода-доступа* — это *блок*, определяющий операторы, выполняемые при вызове соответствующего кода доступа.

Код получения (get) соответствует методу без параметров, тип возвращаемого значения которого совпадает с типом свойства. Когда в выражении ссылаются на свойство, за исключением случая присваивания этому свойству, вызывается его код получения, вычисляющий значение свойства (раздел 7.1.1). Тело кода получения должно подчиняться правилам для методов, возвращающих значение, описанным в разделе 10.6.10. В частности, во всех операторах `return`, расположенных в теле кода получения, должны быть указаны выражения, неявно преобразуемые в тип свойства. Кроме того, конечная точка кода получения должна быть недостижима.

Код установки соответствует методу, принимающему один параметр-значение типа свойства и возвращающему `void`. Неявный параметр кода установки всегда называется `value`. Когда свойство находится слева в операции присваивания (раздел 7.17) или является операндом в операциях `++` или `--` (разделы 7.6.9, 7.7.5), вызывается код установки с аргументом (значение которого равно выражению, находящемуся справа от присваивания, или операнду операции `++` или `--`), предоставляющим новое значение (раздел 7.17.1). Тело кода установки должно подчиняться правилам для методов, возвращающих `void`, описанным в разделе 10.6.10. В частности, в операторах `return`, расположенных в теле кода установки, не допускается указывать выражения. Так как коду установки неявно передается параметр с именем `value`, использование такого имени в объявлении локальной переменной или константы в коде установки приводит к ошибке компиляции.

В зависимости от наличия или отсутствия кодов получения и установки свойства классифицируются следующим образом:

- Свойство, содержащее оба кода доступа, называется свойством, **доступным для чтения и записи**.

- Свойство, содержащее только код получения, называется свойством **только для чтения**. Если такое свойство находится в левой части операции присваивания, возникает ошибка компиляции.
- Свойство, содержащее только код установки, называется свойством **только для записи**. Кроме случая, когда это свойство находится в левой части операции присваивания, попытка сослаться на него в выражении приведет к ошибке компиляции.

Пример:

```
public class Button: Control
{
   private string caption;

   public string Caption {
      get {
          return caption;
      }

      set {
         if (caption != value) {
            caption = value;
            Repaint();
         }
      }
   }

   public override void Paint(Graphics g, Rectangle r) {
       // Здесь расположен код для отрисовки
   }
}
```

В этом примере элемент управления `Button` объявляет открытое свойство `Caption`. Код получения этого свойства возвращает строку, хранящуюся в закрытом поле `caption`. Код установки проверяет, отличается ли новое значение от текущего, и если да, сохраняет новое значение и перерисовывает элемент управления. Свойства часто следуют приведенному выше образцу: код получения просто возвращает значение, хранящееся в закрытом поле, а код установки изменяет значение этого поля и затем выполняет дополнительные действия, необходимые для полного обновления состояния объекта.

Ниже приведен пример использования свойства `Caption` объявленного выше класса `Button`:

```
Button okButton = new Button();
okButton.Caption = "OK";            // Вызывает код установки
string s = okButton.Caption;        // Вызывает код получения
```

Здесь код установки вызывается путем присваивания значения свойству, а код получения — обращением к свойству в выражении.

Коды получения и установки свойства не являются отдельными элементами и объявить их по отдельности невозможно. По существу, невозможно объявить разные виды доступа для двух кодов доступа свойства, доступного для чтения и записи. Пример:

```
class A
{
   private string name;

   public string Name {            // Ошибка: повторяющееся имя элемента
       get { return name; }
   }

   public string Name {            // Ошибка: повторяющееся имя элемента
       set { name = value; }
   }
}
```

Этот пример не объявляет одно свойство, доступное для чтения и записи. Напротив, он объявляет два свойства с одинаковыми именами, одно только для чтения, а другое только для записи. Поскольку два элемента, объявленные в одном классе, не могут иметь одинаковые имена, этот пример приведет к ошибке компиляции.

Когда производный класс объявляет свойство с тем же именем, что и унаследованное, новое свойство скрывает унаследованное как в плане чтения, так и в плане записи.

Пример:

```
class A
{
   public int P {
       set {...}
   }
}

class B: A
{
   new public int P {
       get {...}
   }
}
```

Здесь свойство `P` в `B` скрывает свойство `P` в `A` и при чтении, и при записи. Следовательно в выражениях:

```
B b = new B();
b.P = 1                // Ошибка: B.P доступно только для чтения
((A)b).P = 1;          // Все в порядке: ссылается на A.P
```

Присваивание значения свойству `b.P` приводит к ошибке компиляции, поскольку доступное только для чтения свойство `P` в `B` скрывает доступное только для записи свойство `P` в `A`. Однако заметьте, что для доступа к скрытому свойству `P` можно воспользоваться приведением типов.

В отличие от полей, объявленных как `public`, свойства обеспечивают разделение между внутренним состоянием объекта и его открытым интерфейсом. Рассмотрим следующий пример:

```
class Label
{
   private int x, y;
   private string caption;
```

продолжение ⇨

```
      public Label(int x, int y, string caption) {
         this.x = x;
         this.y = y;
         this.caption = caption;
   }

   public int X {
       get { return x; }
   }

   public int Y {
       get { return y; }
   }

   public Point Location {
       get { return new Point(x, y); }
   }

   public string Caption {
       get { return caption; }
   }
}
```

Здесь класс `Label` использует два поля типа `int` — `x` и `y` — для хранения собственной позиции. Снаружи позиция представлена свойствами `X` и `Y`, а также свойством `Location` типа `Point`. Если в будущей версии `Label` станет более удобным использовать тип `Point` для внутреннего хранения позиции, это изменение можно будет сделать, не меняя открытый интерфейс класса:

```
class Label
{
   private Point location;
   private string caption;

   public Label(int x, int y, string caption) {
      this.location = new Point(x, y);
      this.caption = caption;
   }

   public int X {
       get { return location.x; }
   }

   public int Y {
       get { return location.y; }
   }

   public Point Location {
       get { return location; }
   }

   public string Caption {
       get { return caption; }
   }
}
```

Если бы x и y изначально были полями, объявленными как public readonly, сделать такое изменение в классе Label было бы невозможным.

Обеспечение доступа к состоянию посредством свойств не обязательно менее эффективно, чем непосредственный доступ к полям. В частности, если свойство является невиртуальным и содержит незначительное количество кода, среда выполнения может заменить вызовы кодов доступа их фактическим кодом. Этот процесс известен как **встраивание** и делает доступ к свойству таким же эффективным, как и доступ к полю, при этом сохраняя повышенную гибкость свойств.

Так как вызов кода получения концептуально эквивалентен чтению значения поля, наличие у этих кодов видимых побочных эффектов считается плохим стилем программирования.

Пример:

```
class Counter
{
   private int next;
   public int Next {
       get { return next++; }
   }
}
```

Здесь значение свойства Next зависит от количества предшествующих доступов. Следовательно, доступ к свойству создает видимый побочный эффект, и вместо свойства здесь следует реализовать метод.

Договоренность «об отсутствии побочных эффектов» для кодов получения не означает, что они всегда должны просто возвращать значения, хранящиеся в полях. На самом деле коды получения часто вычисляют значение свойства, осуществляя доступ к нескольким полям или вызывая методы. Однако правильно спроектированный код получения не выполняет действий, приводящих к видимым изменениям в состоянии объекта.

Свойства можно использовать для того, чтобы отложить инициализацию ресурса до момента первого обращения к нему.

Пример:

```
using System.IO;

public class Console
{
   private static TextReader reader;
   private static TextWriter writer;
   private static TextWriter error;

   public static TextReader In {
      get {
         if (reader == null) {
             reader = new StreamReader(Console.OpenStandardInput());
         }
         return reader;
      }
   }
```

продолжение ⇗

```
   public static TextWriter Out {
      get {
         if (writer == null) {
             writer = new StreamWriter(Console.OpenStandardOutput());
         }
         return writer;
      }
   }

   public static TextWriter Error {
      get {
         if (error == null) {
             error = new StreamWriter(Console.OpenStandardError());
         }
         return error;
      }
   }
}
```

В этом примере класс `Console` содержит три свойства: `In`, `Out` и `Error` — которые представляют собой стандартные устройства ввода, вывода и вывода ошибок соответственно. Предоставив доступ к этим элементам через свойства, класс `Console` может отложить их инициализацию до момента их фактического использования. Например, первое обращение к свойству `Out` может выглядеть следующим образом:

```
Console.Out.WriteLine("здравствуй, мир");
```

В этом случае создается соответствующий экземпляр `TextWriter` для устройства вывода. Но если приложение не использует свойства `In` и `Error`, объекты для этих устройств не создаются.

10.7.3. Автоматически реализуемые свойства

> **ДЖОН СКИТ**
>
> В раннем черновом варианте спецификации C# 3.0 эти свойства назывались «автоматическими свойствами». Данное название существовало достаточно долго, чтобы успеть закрепиться, и теперь используется сообществом гораздо чаще, чем полное.

Когда свойство определено как автоматически реализуемое, для него автоматически становится доступным скрытое вспомогательное поле и реализуются коды доступа для чтения этого поля и записи в него.

Пример:

```
public class Point {
    public int X { get; set; }        // Автоматически реализуемое поле
    public int Y { get; set; }        // Автоматически реализуемое поле
}
```

Приведенный пример эквивалентен следующему:

```
public class Point {
   private int x;
   private int y;
   public int X { get { return x; } set { x = value; } }
   public int Y { get { return y; } set { y = value; } }
}
```

Поскольку вспомогательное поле недоступно, его значение можно прочитать и записать только через коды доступа свойства даже в пределах содержащего его типа. Это означает, что автоматически реализуемые свойства, доступные только для чтения или только для записи, не имеют смысла и поэтому запрещены. Для каждого кода доступа можно указать отдельный вид доступа. Таким образом, доступное только для чтения свойство и соответствующее ему закрытое вспомогательное поле можно имитировать следующим образом:

```
public class ReadOnlyPoint {
   public int X { get; private set; }
   public int Y { get; private set; }
   public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

БИЛЛ ВАГНЕР

Хотя свойства доступны только для чтения, соответствующая вспомогательная область памяти доступна также и для записи. Код внутри `ReadOnlyPoint` не обязан подчиняться правилам неизменяемости.

ДЖОН СКИТ

Хотя поведение приведенного кода имитирует свойство, доступное только для чтения остальным типам, оно не эквивалентно свойству (или на самом деле вспомогательному полю), в действительности доступному только для чтения. Я был бы рад возможности определять свойства только для чтения, возможно, следующим образом:

```
public int X { get; readonly set; }
```

Этот код мог бы создавать поле только для чтения; использование кода установки допускалось бы только в конструкторе, где оно компилировалось бы в непосредственную запись во вспомогательное поле. К сожалению, в данный момент создание действительно неизменяемого типа требует большего количества кода, чем создание изменяемого.

ПИТЕР СЕСТОФТ

Автоматически реализуемые свойства прекрасны, потому что свойства можно использовать в инициализаторах объекта (раздел 7.6.10.2). Инициализаторы объекта особенно хорошо подходят для традиционных значений, таких как пары и тройки координат, комплексные числа и другие — то есть структурных значений. Однако в общем случае мы предпочитаем, чтобы поля структур были неизменяемы, поскольку они копируются в процессе присваивания и передачи в качестве параметров. По этой причине я полностью поддерживаю желание Джона использовать краткую нотацию для полей, доступных только для чтения и имеющих автоматически реализуемые коды доступа, при условии, что код установки можно будет использовать в конструкторе *и в инициализаторах объекта*.

Это ограничение также означает, что явное присваивание структурных типов с автоматически реализуемыми свойствами можно осуществить только с помощью стандартного конструктора структуры, поскольку присваивание самому свойству требует, чтобы структура была явно присвоена. Это означает, что конструкторы, определенные пользователем, должны вызывать конструктор по умолчанию.

10.7.4. Доступность

Если код доступа содержит *модификатор-кода-доступа*, область доступности (раздел 3.5.2) кода доступа определяется с помощью объявления вида доступа в *модификаторе-кода-доступа*. Если такой модификатор отсутствует, область доступности определяется с помощью объявления вида доступа свойства или индексатора.

Наличие *модификатора-кода-доступа* никогда не влияет на поиск элемента (раздел 7.3) или разрешение перегрузки (раздел 7.5.3). Модификаторы свойства или индексатора всегда определяют подходящие свойство или индексатор вне зависимости от контекста доступа.

Как только были выбраны конкретные свойство или индексатор, для определения корректности их использования задействуются области доступности конкретных кодов доступа:

- Если свойство используется как значение (раздел 7.1.1), должен существовать и быть доступным код получения.
- Если свойство используется в левой части простого присваивания (раздел 7.17.1), должен существовать и быть доступным код установки.
- Если свойство используется в левой части составного присваивания (раздел 7.17.2) или в операциях `++` или `--` (разделы 7.5.9, 7.6.5), должны существовать и быть доступными оба кода доступа.

В следующем примере свойство `A.Text` скрыто свойством `B.Text` даже в тех контекстах, в которых вызывается только код установки. Свойство `B.Count`, напротив, недоступно классу `M`, поэтому вместо него используется доступное свойство `A.Count`.

```
class A
{
   public string Text {
      get { return "привет"; }
      set { }
   }

   public int Count {
      get { return 5; }
      set { }
   }
}

class B: A
{
   private string text = "пока";
   private int count = 0;
```

```
   new public string Text {
      get { return text; }
      protected set { text = value; }
   }

   new protected int Count {
      get { return count; }
      set { count = value; }
   }
}

class M
{
   static void Main() {
      B b = new B();
      b.Count = 12;               // Вызывает код установки для A.Count
      int i = b.Count;            // Вызывает код получения для A.Count
      b.Text = "здорово!";        // Ошибка: код установки для B.Text
                                  // недоступен
      string s = b.Text;          // Вызывает код получения для B.Text
   }
}
```

Код доступа, использующийся для реализации интерфейса, не может содержать *модификатор-кода-доступа*. Если для реализации интерфейса используется только один код доступа, объявление второго может содержать *модификатор-кода-доступа*:

```
public interface I
{
    string Prop { get; }
}

public class C: I
{
   public Prop {
      get { return "Апрель"; }    // Здесь не может быть модификатора
      internal set {...}          // Все в порядке — у I.Prop
                                  // нет кода установки
   }
}
```

10.7.5. Виртуальные, бесплодные, переопределенные и абстрактные коды доступа

Объявление свойства как `virtual` указывает на то, что коды доступа этого свойства являются виртуальными. Модификатор `virtual` применяется к обоим кодам доступа свойства, доступного для чтения и записи — невозможно сделать виртуальным только один код доступа такого свойства.

Объявление свойства как `abstract` указывает на то, что коды доступа этого свойства являются виртуальными, но не предоставляет их фактической реализации. Вместо этого неабстрактные производные классы обязаны предоставить соб-

ственную реализацию кодов доступа, переопределив свойство. Так как код доступа в объявлении абстрактного свойства не предоставляет фактической реализации, *тело-кода-доступа* состоит просто из точки с запятой.

Объявление свойства, содержащее одновременно модификаторы `abstract` и `override`, указывает на то, что свойство является абстрактным и переопределяет базовое свойство. Коды доступа такого свойства также абстрактны.

Объявления абстрактных свойств допустимы только в абстрактных классах (раздел 10.1.1.1). Производный класс может переопределить коды доступа унаследованного виртуального свойства с помощью объявления свойства, содержащего директиву `override`. Оно называется **переопределяющим объявлением свойства**. Такое объявление не объявляет новое свойство. Вместо этого оно просто уточняет реализации кодов доступа существующего виртуального свойства.

Переопределяющее объявление свойства должно иметь точно такие же модификаторы доступа, тип и имя, что и унаследованное свойство. Если последнее содержит только один код доступа (то есть доступно только для чтения или только для записи), переопределенное свойство должно содержать только этот код доступа. Если унаследованное свойство содержит оба кода доступа (то есть доступно для чтения и записи), переопределенное свойство может содержать либо один, либо оба кода доступа.

Переопределяющее объявление свойства может содержать модификатор `sealed`. Использование этого модификатора лишает производный класс возможности повторно переопределить свойство. Коды доступа бесплодного свойства также являются бесплодными.

За исключением различий в синтаксисе объявления и вызова виртуальные, бесплодные, переопределенные и абстрактные коды доступа ведут себя в точности так же, как виртуальные, бесплодные, переопределенные и абстрактные методы. Точнее, правила, описанные в разделах 10.6.3—10.6.6, применяются так же, как если бы коды доступа были методами в следующей форме:

- Код получения соответствует методу без параметров, возвращающему значение с типом свойства и имеющему те же модификаторы, что и содержащее его свойство.
- Код установки соответствует методу с одним параметром-значением типа свойства, возвращающему `void` и имеющему те же модификаторы, что и содержащее его свойство.

Пример:

```
abstract class A
{
   int y;

   public virtual int X {
       get { return 0; }
   }

   public virtual int Y {
      get { return y; }
      set { y = value; }
   }
```

```
    public abstract int Z { get; set; }
}
```

Здесь X является виртуальным свойством только для чтения, Y — виртуальным свойством, доступным для чтения и записи, а Z — абстрактным свойством, доступным для чтения и записи. Поскольку Z — абстрактное свойство, содержащий его класс A также должен быть объявлен как абстрактный.

Ниже показан класс, унаследованный от A:

```
class B: A
{
   int z;

   public override int X {
       get { return base.X + 1; }
   }

   public override int Y {
       set { base.Y = value < 0? 0: value; }
   }

   public override int Z {
      get { return z; }
      set { z = value; }
   }
}
```

Здесь объявления X, Y и Z представляют собой переопределяющие объявления свойств. Модификаторы, тип и имя в объявлении каждого свойства в точности совпадают с используемыми в объявлении унаследованного свойства. Код получения в X и код установки в Y используют ключевое слово base для доступа к унаследованным кодам доступа. Объявление Z переопределяет оба абстрактных кода доступа — таким образом, в B не остается абстрактных функциональных элементов, поэтому данный класс может быть объявлен неабстрактным.

Когда свойство объявлено как override, любые переопределяемые коды доступа должны быть доступны в переопределяющем коде. Кроме этого, объявления вида доступа свойства или индексатора и кодов доступа должны совпадать с соответствующими объявлениями переопределяемых элемента и кодов доступа. Пример:

```
public class D: B
{
   public override int P {
      protected set {...}
      get {...}
   }
}

public class B
{
   public virtual int P {
      protected set {...}        // Здесь требуется указать protected
      get {...}                  // Здесь не должно быть модификатора
   }
}
```

10.8. События

Событие — это элемент, позволяющий объекту или классу предоставлять уведомления. Клиенты могут прикреплять к событиям выполняемый код, используя **обработчики событий**.

События объявляются с помощью *объявлений-событий*:

```
объявление-события:
   атрибуты_opt   модификаторы-события_opt   event   тип
      описатели-переменных   ;
   атрибуты_opt   модификаторы-события_opt   event   тип   имя-элемента
      {   объявления-кодов-доступа-события   }

модификаторы-события:
   модификатор-события
   модификаторы-события   модификатор-события

модификатор-события:
   new
   public
   protected
   internal
   private
   static
   virtual
   sealed
   override
   abstract
   extern

объявления-кодов-доступа-события:
   объявление-кода-доступа-add   объявление-кода-доступа-remove
   объявление-кода-доступа-remove   объявление-кода-доступа-add

объявление-кода-доступа-add:
   атрибуты_opt   add   блок

объявление-кода-доступа-remove:
   атрибуты_opt   remove   блок
```

Объявление-события может содержать набор *атрибутов* (раздел 17) и корректную комбинацию четырех модификаторов доступа (раздел 10.3.5) и модификаторов `new` (раздел 10.3.4), `static` (раздел 10.6.2), `virtual` (раздел 10.6.3), `override` (раздел 10.6.4), `sealed` (раздел 10.6.5), `abstract` (раздел 10.6.6) и `extern` (раздел 10.6.7).

В том, что касается корректных комбинаций модификаторов, объявления событий подчиняются тем же правилам, что и объявления методов (раздел 10.6).

Тип в объявлении события должен являться *делегатом* (раздел 4.2), и этот *делегат* должен иметь не более строгий вид доступа, чем само событие (раздел 3.5.4).

Объявление события может содержать *объявления-кодов-доступа-события*. Однако если их нет, для событий, не являющихся внешними или абстрактными, компилятор подставляет их автоматически (раздел 10.8.1); для внешних событий коды доступа предоставляются во внешнем коде.

Объявление события, в котором отсутствуют *объявления-кодов-доступа-события*, определяет одно или несколько событий — по одному для каждого *описателя-переменной*. Атрибуты и модификаторы применяются ко всем элементам, объявленным таким *объявлением-события*.

Если *объявление-события* одновременно содержит модификатор `abstract` и заключенные в фигурные скобки *объявления-кодов-доступа-события*, возникает ошибка компиляции.

Когда объявление события содержит модификатор `extern`, событие называется **внешним событием**. Поскольку объявление внешнего события не предоставляет фактической реализации, указание одновременно модификатора `extern` и *объявлений-кодов-доступа-события* приводит к ошибке.

Событие можно использовать в качестве левого операнда операций `+=` и `-=` (раздел 7.17.3). Эти операции используются для добавления обработчиков события и их удаления соответственно, а модификаторы доступа события определяют контексты, в которых допускается использование этих операций.

Поскольку за пределами типа, объявляющего событие, для последнего допускаются только операции `+=` и `-=`, внешний код может добавлять или удалять обработчики события, но не может каким-либо другим способом получать или модифицировать лежащий в основе список обработчиков.

В операции, имеющей вид `x += y` или `x -= y`, если `x` — событие, а обращение к нему производится за пределами типа, содержащего объявление `x`, результат операции имеет тип `void` (в противоположность результату, имеющему после присваивания тип и значение `x`). Это правило лишает внешний код возможности косвенно анализировать связанный с событием делегат.

Следующий пример демонстрирует, как события прикрепляются к экземплярам класса `Button`:

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form

{
   Button OkButton;
   Button CancelButton;

   public LoginDialog() {
      OkButton = new Button(...);
      OkButton.Click += new EventHandler(OkButtonClick);
      CancelButton = new Button(...);
      CancelButton.Click += new EventHandler(CancelButtonClick);
   }

   void OkButtonClick(object sender, EventArgs e) {
       // Обработка события OkButton.Click
   }
```

продолжение ☞

```
    void CancelButtonClick(object sender, EventArgs e) {
        // Обработка события CancelButton.Click
    }
}
```

Здесь конструктор экземпляра `LoginDialog` создает два экземпляра `Button` и прикрепляет обработчики к событиям `Click`.

> **ХРИСТИАН НЕЙГЕЛ**
>
> Неправильное использование событий часто приводит к утечкам памяти. Если клиентские объекты прикрепляются к событиям, но не открепляются от них, а ссылка на клиентский объект больше не используется, этот объект все равно не может быть уничтожен сборщиком мусора, поскольку на него все еще ссылается объект, объявивший событие. Такой ситуации можно избежать, (1) открепляя события, когда клиентский объект перестает использоваться, (2) предоставив собственную реализацию кодов доступа `add` и `remove`, использующую класс `WeakReference` для хранения делегата, или (3) задействовав шаблон «Слабое Событие» (Weak Event), который используется в WPF с интерфейсом `IWeakEventListener`.

> **ДЖОН СКИТ**
>
> Я склонен думать (и давать разъяснения) о событиях как о чем-то, похожем на свойства: они представляют собой пару методов (`add` и `remove`), способ вызова которых с помощью сокращенных обозначений (`+=` и `-=`) известен компилятору. События более ограничены, чем свойства, в том плане, что типом события всегда должен быть тип делегата и *всегда* должны присутствовать методы `add` и `remove`; не существует такого понятия, как событие, «доступное только для добавления», хотя свойства могут быть доступны только для чтения. Не считая этих пунктов, наблюдается сильное сходство — и тем не менее свойства обычно очень доступны для понимания, а вот события часто приводят в замешательство. Я подозреваю, что причиной проблемы являются события, подобные полям.

10.8.1. События, подобные полям

В коде класса или структуры, содержащих объявление события, некоторые события могут использоваться как поля. Для такого использования событие не должно быть абстрактным или внешним и не должно явно содержать *объявления-кодов-доступа-события*. Данное событие можно использовать в любых контекстах, в которых допускается использование поля. Поле содержит делегат (раздел 15), который ссылается на список обработчиков, добавленных к этому событию. Если обработчики не добавлялись, поле содержит `null`.

> **ДЖОН СКИТ**
>
> Если бы автоматически реализуемые свойства присутствовали в C# с самого начала, я думаю, было бы более разумным назвать события, подобные полям, «автоматически реализуемыми событиями» с синтаксисом, похожим на следующий:

```
public event EventHandler Click { add; remove; }
```
Такой подход создавал бы меньше путаницы относительно того, чем на самом деле являются события.

Пример:

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
   public event EventHandler Click;

   protected void OnClick(EventArgs e) {
       if (Click != null) Click(this, e);
   }

   public void Reset() {
       Click = null;
   }
}
```

Здесь `Click` используется как поле в классе `Button`. Как показано в примере, можно проверять значение поля, изменять его и использовать в выражениях вызова делегата. Метод `OnClick` в классе `Button` «генерирует» событие `Click`. Понятие генерации события в точности эквивалентно вызову делегата, представленного событием — таким образом, для генерации событий не существует специальных языковых конструкций. Заметьте, что вызову делегата предшествует проверка того, что он не является нулевым.

За пределами объявления класса `Button` элемент `Click` можно использовать только в левой части операций `+=` и `-=`, как в примере

```
b.Click += new EventHandler(...);
```

который добавляет делегат к списку вызова события `Click`, и в примере

```
b.Click -= new EventHandler(...);
```

который удаляет делегат из списка вызова события `Click`.

При компиляции подобного полю события компилятор автоматически создает область памяти для хранения делегата, а также коды доступа события, которые добавляют обработчики к полю делегата или удаляют их. Операции добавления и удаления безопасны с точки зрения потоков и могут (но не обязаны) выполняться при удержании блокировки (раздел 8.12) охватывающего объекта для события экземпляра или объекта типа (раздел 7.6.10.6) для статического события.

Таким образом, объявление события экземпляра в форме

```
class X
{
    public event D Ev;
}
```

будет скомпилировано во что-то, похожее на:

```
class X
{
   private D __Ev;            // Поле для хранения делегата
```

продолжение ⇨

```
    public event D Ev {
        add {
                /* Добавление делегата потокобезопасным способом */
        }
        remove {
                /* Удаление делегата потокобезопасным способом */
        }
    }
}
```

Внутри класса X обращения к Ev в левой части операций += и -= приводят к вызовам кодов доступа add и remove. Все остальные ссылки на Ev компилируются так, чтобы вместо этого ссылаться на скрытое поле __Ev. Имя «__Ev» выбрано произвольно; скрытое поле может иметь любое имя или не иметь его вовсе.

10.8.2. Коды доступа событий

Объявления событий обычно не содержат *объявлений-кодов-доступа-событий*, как в приведенном выше примере Button. Одним из случаев, когда это может потребоваться, является ситуация, в которой стоимость хранения каждого события в виде одного поля неприемлема. В таких случаях класс может содержать *объявления-кодов-доступа-событий* и использовать закрытый механизм для хранения списка обработчиков.

Объявления-кодов-доступа-событий для события определяют выполняемые операторы, связанные с добавлением и удалением обработчиков.

Объявления кодов доступа состоят из *объявления-кода-доступа-add* и *объявления-кода-доступа-remove*. Каждое объявление кода доступа состоит из лексемы add или remove, за которой следует *блок*. *Блок*, ассоциированный с *объявлением-кода-доступа-add*, определяет операторы, выполняемые при добавлении обработчика, а *блок*, ассоциированный с *объявлением-кода-доступа-remove*, определяет операторы, выполняемые при удалении обработчика события.

Каждое *объявление-кода-доступа-add* и *объявление-кода-доступа-remove* соответствует методу с одним параметром-значением, имеющим тип события, возвращающему void. Неявный параметр кода доступа события называется value. Когда событие используется в присваивании событию, используется соответствующий код доступа: для операции присваивания += используется код доступа add, а для операции -= — код доступа remove. В обоих случаях правый операнд операции используется в качестве аргумента для кода доступа. Блок в *объявлении-кода-доступа-add* и *объявлении-кода-доступа-remove* должен подчиняться правилам для методов, возвращающих void, описанным в разделе 10.6.10. В частности, в операторах return внутри этого блока нельзя указывать выражения.

Так как код доступа события неявно содержит параметр с именем value, объявление в нем локальной переменной или константы с таким же именем приведет к ошибке компиляции.

Пример:

```
class Control: Component
{
   // Уникальные ключи для событий
   static readonly object mouseDownEventKey = new object();
   static readonly object mouseUpEventKey = new object();

   // Возвращает обработчик события, ассоциированный с ключом
   protected Delegate GetEventHandler(object key) {...}

   // Добавляет обработчик события, ассоциированный с ключом
   protected void AddEventHandler(object key, Delegate handler) {...}

   // Удаляет обработчик события, ассоциированный с ключом
   protected void RemoveEventHandler(object key, Delegate handler) {...}

   // Событие MouseDown
   public event MouseEventHandler MouseDown {
      add { AddEventHandler(mouseDownEventKey, value); }
      remove { RemoveEventHandler(mouseDownEventKey, value); }
   }

   // Событие MouseUp
   public event MouseEventHandler MouseUp {
      add { AddEventHandler(mouseUpEventKey, value); }
      remove { RemoveEventHandler(mouseUpEventKey, value); }
   }

   // Вызов события MouseUp
   protected void OnMouseUp(MouseEventArgs args) {
      MouseEventHandler handler;
      handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
      if (handler != null)
              handler(this, args);
   }
}
```

В этом примере класс `Control` реализует внутренний механизм для хранения событий. Метод `AddEventHandler` ассоциирует значение делегата с ключом, метод `GetEventHandler` возвращает текущий делегат, ассоциированный с ключом, а метод `RemoveEventHandler` удаляет делегат, являющийся обработчиком для указанного события. Предположительно, лежащий в основе механизм хранения спроектирован таким образом, что ассоциирование с ключом делегата со значением `null` не приводит к накладным расходам, следовательно, необработанные события не расходуют память.

10.8.3. Статические события и события экземпляра

Когда объявление события содержит модификатор `static`, событие называется **статическим событием**. Если модификатор `static` отсутствует, событие называется **событием экземпляра**.

Статическое событие не связано с конкретным экземпляром, и попытка обратиться к `this` в кодах доступа статического события приведет к ошибке компиляции.

Событие экземпляра связано с конкретным экземпляром класса, и в кодах доступа события на этот экземпляр можно ссылаться с помощью `this` (раздел 7.6.7).

Когда к событию обращаются с помощью *доступа-к-элементу* (раздел 7.6.4) в форме `E.M` и `M` — статическое событие, `E` должен обозначать тип, содержащий `M`; если `M` — событие экземпляра, `E` должен обозначать экземпляр типа, содержащего `M`.

Более подробно различия между статическими элементами и элементами экземпляра рассматриваются в разделе 10.3.7.

10.8.4. Виртуальные, бесплодные, переопределенные и абстрактные коды доступа

Объявление события как `virtual` указывает на то, что коды доступа этого события являются виртуальными. Модификатор `virtual` применяется к обоим кодам доступа события.

Объявление события как `abstract` указывает на то, что коды доступа события являются виртуальными, но не предоставляет их фактической реализации. Вместо этого неабстрактные производные классы обязаны предоставить собственную реализацию кодов доступа, переопределив событие. Так как объявление абстрактного события не предоставляет фактической реализации, оно не может содержать заключенных в фигурные скобки *объявлений-кодов-доступа-события*.

Объявление события, содержащее одновременно модификаторы `abstract` и `override`, указывает на то, что событие является абстрактным и переопределяет базовое событие. Коды доступа такого события также абстрактны.

Объявления абстрактных событий допустимы только в абстрактных классах (раздел 10.1.1.1).

Производный класс может переопределить коды доступа унаследованного виртуального события с помощью объявления события, содержащего модификатор `override`. Оно называется **переопределяющим объявлением события**. Такое объявление не объявляет новое событие, а просто уточняет реализации кодов доступа существующего виртуального события.

Переопределяющее объявление события должно иметь точно такие же модификаторы доступа, тип и имя, что и унаследованное событие.

Переопределяющее объявление события может содержать модификатор `sealed`. Использование этого модификатора лишает унаследованный класс возможности повторно переопределить событие. Коды доступа бесплодного события также являются бесплодными.

Если переопределяющее объявление события содержит модификатор `new`, возникает ошибка компиляции.

За исключением различий в синтаксисе объявления и вызова виртуальные, бесплодные, переопределенные и абстрактные коды доступа ведут себя в точности

так же, как виртуальные, бесплодные, переопределенные и абстрактные методы. Точнее, правила, описанные в разделах 10.6.3—10.6.6, применяются так же, как если бы коды доступа были методами в соответствующей форме. Каждый код доступа соответствует методу с одним параметром-значением типа события, возвращающему `void` и имеющему те же модификаторы, что и содержащее его событие.

10.9. Индексаторы

Индексатор — это элемент, позволяющий обращаться к объекту по индексу тем же способом, что и к массиву. Индексаторы объявляются с помощью *объявлений-индексаторов*:

```
объявление-индексатора:
   атрибуты_opt   модификаторы-индексатора_opt   описатель-индексатора
      {   объявления-кодов-доступа   }

модификаторы-индексатора:
   модификатор-индексатора
   модификаторы-индексатора   модификатор-индексатора

модификатор-индексатора:
   new
   public
   protected
   internal
   private
   virtual
   sealed
   override
   abstract
   extern

описатель-индексатора:
   тип   this   [   список-формальных-параметров   ]
   тип   интерфейс   .   this   [   список-формальных-параметров   ]
```

Объявление-индексатора может содержать набор *атрибутов* (раздел 17) и корректную комбинацию четырех модификаторов доступа (раздел 10.3.5) и модификаторов `new` (раздел 10.3.4), `virtual` (раздел 10.6.3), `override` (раздел 10.6.4), `sealed` (раздел 10.6.5), `abstract` (раздел 10.6.6) и `extern` (раздел 10.6.7).

В том, что касается корректных комбинаций модификаторов, объявления индексаторов подчиняются тем же правилам, что и объявления методов (раздел 10.6), за исключением того, что в объявлении индексатора недопустим модификатор `static`.

Модификаторы `virtual`, `override` и `abstract` являются взаимно исключающими, кроме одного случая. Модификаторы `abstract` и `override` могут использоваться вместе, чтобы абстрактный индексатор мог переопределить виртуальный.

Тип в объявлении индексатора определяет тип элемента индексатора, вводимого объявлением. Если индексатор не является явной реализацией элемента интерфейса,

за *типом* следует ключевое слово `this`. Для явной реализации элемента интерфейса за *типом* следуют *интерфейс*, символ «`.`» и ключевое слово `this`. В отличие от остальных элементов индексаторы не имеют определяемых пользователем имен.

Список-формальных-параметров определяет параметры индексатора. Список формальных параметров индексатора соответствует аналогичному списку метода (раздел 10.6.1), за исключением того, что в нем требуется указать хотя бы один параметр и не допускается использование модификаторов `ref` и `out`.

Тип индексатора и каждый из типов, перечисленных в *списке-формальных-параметров*, должны иметь не более строгий вид доступа, чем сам индексатор (раздел 3.5.4).

Объявления-кодов-доступа (раздел 10.7.2), которые должны находиться внутри лексем «`{`» и «`}`», объявляют коды доступа индексатора. Коды доступа определяют выполняемые операторы, ассоциированные с чтением и записью элементов-индексаторов.

Несмотря на то что синтаксис доступа к элементу индексатора аналогичен синтаксису доступа к элементу массива, элемент индексатора не классифицируется как переменная. Следовательно, невозможно передать элемент индексатора в качестве аргумента `ref` или `out`.

Список формальных параметров индексатора определяет сигнатуру (раздел 3.6) индексатора. Точнее, сигнатура индексатора состоит из количества и типов его формальных параметров. Тип элемента и имена формальных параметров к ней не относятся.

Сигнатура индексатора должна отличаться от сигнатур всех остальных индексаторов, объявленных в том же классе.

Индексаторы концептуально очень похожи на свойства, но имеют следующие отличия:

- Свойство определяется своим именем, а индексатор — своей сигнатурой.
- К свойству обращаются с помощью *простого-имени* (раздел 7.6.2) или *доступа-к-элементу* (раздел 7.6.4), а к индексатору — с помощью *доступа-к-элементу-массива* (раздел 7.6.6.2).
- Свойство может быть статическим элементом, а индексатор всегда является элементом экземпляра.
- Код получения у свойства соответствует методу без параметров, а у индексатора он соответствует методу с тем же списком формальных параметров, что и у индексатора.
- Код установки у свойства соответствует методу с одним параметром с именем `value`, а у индексатора он соответствует методу с тем же списком формальных параметров, что и у индексатора, а также дополнительным параметром с именем `value`.
- Если в коде доступа индексатора объявлена локальная переменная с тем же именем, что и параметр индексатора, возникает ошибка компиляции.
- В переопределяющем объявлении свойства доступ к унаследованному свойству можно получить, используя синтаксис `base.P`, где `P` — имя свойства. В пере-

определяющем объявлении индексатора доступ к унаследованному индексатору можно получить с помощью синтаксиса `base[E]`, где `E` — список выражений, разделенных запятыми.

Не считая этих отличий, все правила, приведенные в разделе 10.7.2 и разделе 10.7.3, применяются к кодам доступа индексаторов так же, как и к кодам доступа свойств.

> **ДЖОН СКИТ**
>
> Мне кажется странным, что события могут быть статическими (но почти никогда не бывают), а индексаторы — нет. Например, метод `Encoding.GetEncoding(string)` разумно было бы представить в виде статического индексатора. На самом деле я думаю, что статические индексаторы в первую очередь использовались бы как элементы-фабрики для типа, в котором они объявлены. В большинстве случаев для этих целей прекрасно подходит и метод, но такой запрет кажется немного странным.

Когда объявление индексатора содержит модификатор `extern`, индексатор называется **внешним индексатором**. Поскольку объявление внешнего индексатора не предоставляет фактической реализации, каждое из *объявлений-кодов-доступа* в нем состоит из точки с запятой.

В приведенном ниже примере объявлен класс `BitArray`, реализующий индексатор для доступа к отдельным битам в битовом массиве:

```
using System;

class BitArray
{
   int[] bits;
   int length;

   public BitArray(int length) {
      if (length < 0) throw new ArgumentException();
      bits = new int[((length - 1) >> 5) + 1];
      this.length = length;
   }

   public int Length {
       get { return length; }
   }

   public bool this[int index] {
      get {
         if (index < 0 || index >= length) {
             throw new IndexOutOfRangeException();
         }
         return (bits[index >> 5] & 1 << index) != 0;
      }

      set {
         if (index < 0 || index >= length) {
             throw new IndexOutOfRangeException();
         }
```

продолжение ☞

```
        if (value) {
            bits[index >> 5] |= 1 << index;
        }
        else {
            bits[index >> 5] &= ~(1 << index);
        }
      }
   }
}
```

Экземпляр класса `BitArray` потребляет существенно меньше памяти, чем соответствующий массив `bool[]` (поскольку каждое значение первого занимает только один бит, а последнего — один байт), но допускает использование тех же операций, что и `bool[]`.

Приведенный ниже класс `CountPrimes` использует `BitArray` и классический алгоритм «решето» для вычисления количества простых чисел между 1 и заданным максимальным значением:

```
class CountPrimes
{
   static int Count(int max) {
      BitArray flags = new BitArray(max + 1);
      int count = 1;
      for (int i = 2; i <= max; i++) {
         if (!flags[i]) {
            for (int j = i * 2; j <= max; j += i) flags[j] = true;
            count++;
         }
      }
      return count;
   }

   static void Main(string[] args) {
      int max = int.Parse(args[0]);
      int count = Count(max);
      Console.WriteLine("Найдено {0} простых чисел между 1 и {1}",
                        count, max);
   }
}
```

Заметьте, что синтаксис для доступа к элементам `BitArray` в точности эквивалентен синтаксису для доступа к элементам `bool[]`.

В нижеследующем примере показан класс, реализующий сетку `26x10` и имеющий индексатор с двумя параметрами. Первый параметр должен быть прописной или строчной буквой в диапазоне `A-Z`, а второй — целым числом в диапазоне `0-9`.

```
using System;

class Grid
{
   const int NumRows = 26;
   const int NumCols = 10;

   int[,] cells = new int[NumRows, NumCols];
```

```
   public int this[char c, int col] {
      get {
         c = Char.ToUpper(c);
         if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
         }
         if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
         }
         return cells[c - 'A', col];
      }

      set {
         c = Char.ToUpper(c);
         if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
         }
         if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
         }
         cells[c - 'A', col] = value;
           }
   }
}
```

10.9.1. Перегрузка индексаторов

Правила разрешения перегрузки индексаторов описаны в разделе 7.5.2.

10.10. Операции

Операция — это элемент, определяющий смысл знака операции в выражении, который можно применить к экземплярам класса. Операции объявляются с помощью *объявлений-операций*:

объявление-операции:
   атрибуты$_{opt}$ *модификаторы-операции*   *описатель-операции*   *тело-операции*

модификаторы-операции:
   модификатор-операции
   модификаторы-операции   *модификатор-операции*

модификатор-операции:
   public
   static
   extern

описатель-операции:
   описатель-унарной-операции
   описатель-бинарной-операции
   описатель-операции-преобразования

продолжение ⇗

```
описатель-унарной-операции:
   тип operator перегружаемая-унарная-операция
      ( тип идентификатор )

перегружаемая-унарная-операция: одна из
   + - ! ~ ++ -- true false

описатель-бинарной-операции:
   тип operator перегружаемая-бинарная-операция
      ( тип идентификатор , тип идентификатор )

перегружаемая-бинарная-операция:
   +
   -
   *
   /
   %
   &
   |
   ^
   <<
   сдвиг-вправо
   ==
   !=
   >
   <
   >=
   <=

описатель-операции-преобразования:
   implicit operator тип ( тип идентификатор )
   explicit operator тип ( тип идентификатор )

тело-операции:
   блок
   ;
```

Существуют три категории перегружаемых операций: унарные операции (раздел 10.10.1), бинарные операции (раздел 10.10.2) и операции преобразования (раздел 10.10.3).

Когда объявление операции содержит модификатор `extern`, операция называется **внешней операцией**. Поскольку внешняя операция не предоставляет фактической реализации, *тело-операции* в ней состоит из точки с запятой. Для всех остальных операций *тело-операции* состоит из *блока*, определяющего операторы, выполняемые при вызове операции. *Блок* в операции должен подчиняться правилам для методов, возвращающих значение, описанным в разделе 10.6.10.

Ко всем объявлениям операций применяются следующие правила:

- Объявление операции должно содержать одновременно модификаторы `public` и `static`.
- Параметры операции должны быть параметрами-значениями. Если объявление операции содержит параметры `ref` или `out`, возникает ошибка компиляции.
- Сигнатура операции (разделы 10.10.1—10.10.3) должна отличаться от сигнатур всех остальных операций, объявленных в том же классе.

- Все типы, использующиеся в объявлении операции, должны иметь не более строгий вид доступа, чем сама операция (раздел 3.5.4).
- Если один и тот же модификатор встречается в объявлении операции несколько раз, возникает ошибка компиляции.

Как описано в следующих разделах, на каждую категорию операций налагаются дополнительные ограничения.

Как и другие элементы, операции, объявленные в базовом классе, наследуются производными классами. Поскольку объявления операций требуют, чтобы в сигнатуре операции всегда использовались класс или структура, в которых она объявлена, операция в производном классе не может скрывать операцию, объявленную в базовом классе. Следовательно, использование модификатора `new` в объявлении никогда не требуется и поэтому недопустимо.

Дополнительную информацию об унарных и бинарных операциях можно найти в разделе 7.3.

Дополнительную информацию об операциях преобразования можно найти в разделе 6.4.

10.10.1. Унарные операции

К объявлениям унарных операций применяются следующие правила, где `T` обозначает экземпляр класса или структуры, которые содержат объявление операции:

- Унарная операция `+`, `-`, `!` или `~` должна принимать один параметр типа `T` или `T?` и может возвращать любой тип.
- Унарная операция `++` или `--` должна принимать один параметр типа `T` или `T?` и возвращать тот же самый тип или унаследованный от него тип.
- Унарная операция `true` или `false` должна принимать один параметр типа `T` или `T?` и возвращать тип `bool`.

Сигнатура унарной операции состоит из лексемы операции (`+`, `-`, `!`, `~`, `++`, `--`, `true` или `false`) и типа единственного формального параметра. Тип возвращаемого значения не входит в сигнатуру унарной операции, равно как и имя формального параметра.

Унарные операции `true` и `false` должны объявляться парами. Если в классе объявлена одна из них, но не объявлена вторая, возникает ошибка компиляции. Более подробно операции `true` и `false` описаны в разделах 7.12.2 и 7.20.

Следующий пример демонстрирует реализацию и последующее использование операции `++` для класса, представляющего целочисленный вектор:

```
public class IntVector
{
   public IntVector(int length) {...}

   public int Length {...}                 // Свойство только для чтения

   public int this[int index] {...}        // Индексатор для чтения-записи
```

продолжение ⇗

```
      public static IntVector operator ++(IntVector iv) {
      IntVector temp = new IntVector(iv.Length);
      for (int i = 0; i < iv.Length; i++)
              temp[i] = iv[i] + 1;
      return temp;
   }
}

class Test
{
   static void Main() {
      IntVector iv1 = new IntVector(4);       // Вектор из 4-х нулей
      IntVector iv2;

      iv2 = iv1++;                            // iv2 содержит 4 нуля,
                                              // iv1 содержит 4 единицы
      iv2 = ++iv1;                            // iv2 содержит 4 двойки,
                                              // iv1 содержит 4 двойки
    }
}
```

Заметьте, что метод, реализующий операцию, возвращает значение, получаемое добавлением `1` к операнду, точно так же, как постфиксные операции инкремента и декремента (раздел 7.6.9) и префиксные операции инкремента и декремента (раздел 7.7.5). В отличие от C++ в этом методе не требуется непосредственно изменять значение его операнда. Фактически изменение значения операнда нарушило бы стандартную семантику операции постфиксного инкремента.

10.10.2. Бинарные операции

К объявлениям бинарных операций применяются следующие правила, где `T` обозначает экземпляр класса или структуры, содержащих объявление операции:

- Бинарная операция, не выполняющая сдвиг, должна принимать два параметра, хотя бы один из которых должен иметь тип `T` или `T?`, и может возвращать любой тип.
- Бинарная операция `<<` или `>>` должна принимать два параметра, первый из которых должен иметь тип `T` или `T?`, а второй — `int` или `int?`, и может возвращать любой тип.

Сигнатура бинарной операции состоит из лексемы операции (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=` или `<=`) и типов двух формальных параметров. Тип возвращаемого значения и имена формальных параметров не входят в сигнатуру бинарной операции.

Некоторые бинарные операции требуют попарного объявления. Для каждого объявления одной из парных операций должно присутствовать соответствующее ему объявление второй операции. Два объявления операций соответствуют друг другу, если имеют один и тот же тип возвращаемого значения и одинаковые типы каждого параметра. Следующие операции требуют попарного объявления:

- `operator ==` и `operator !=`
- `operator >` и `operator <`
- `operator >=` и `operator <=`

ЭРИК ЛИППЕРТ

Кажется заманчивой идея реализовать операцию сравнения для класса `Clothing` (одежда), который мог бы представлять `Socks` (носки), `Shoes` (ботинки), `Shirt` (рубашка), `Tie` (галстук) и `Hat` (шляпа), а затем определить сравнение, которое делает `Socks < Shoes`, `Shirt < Tie`, а все остальные предметы равными друг другу. Идея заключается в том, что при сортировке массива этих элементов они расположились бы в удобном порядке — `Socks` всегда бы располагались перед `Shoes`, `Shirts` — перед `Ties`, а все остальное — в произвольном порядке. Но такое сравнение логически несогласованно: если `Hat` равна `Socks`, `Hat` равна `Shoes`, а `Socks` меньше, чем `Shoes`, то с точки зрения логики `Hat` должна быть меньше, чем `Hat`, что бессмысленно. Большинство алгоритмов сортировки написаны в предположении, что операция сравнения определяет *полное упорядочивание*; некоторые алгоритмы завершаются аварийно, выполняются бесконечно или выдают бессмысленные результаты, если в них используется неправильная операция сравнения. Если вы хотите реализовать *сортировку с частичным упорядочиванием*, разумно использовать для этого какой-либо иной механизм, чем перегрузка операций сравнения.

10.10.3. Операции преобразования

Объявление операции преобразования вводит **определенное пользователем преобразование** (раздел 6.4), дополняющее встроенные неявные и явные преобразования.

Объявление операции преобразования, содержащее ключевое слово `implicit`, вводит определенное пользователем неявное преобразование. Неявные преобразования могут использоваться во многих ситуациях, включая вызовы функциональных элементов, выражения приведения типов и присваивания. Подробнее они описаны в разделе 6.1.

БИЛЛ ВАГНЕР

Неявные преобразования всегда должны завершаться успешно и никогда не должны приводить к потерям информации, поскольку они будут вызываться автоматически, без ведома программистов, использующих этот код. По этой же причине при неявных преобразованиях никогда не должны выполняться ресурсоемкие операции.

ДЖОН СКИТ

Как отмечено в разделе 6.12, некоторые неявные преобразования из целочисленных типов к типам `float` или `double` могут приводить к потере информации. Однако такая возможность не должна использоваться как оправдание потери информации в ваших собственных неявных преобразованиях.

Объявление операции преобразования, содержащее ключевое слово `explicit`, вводит определенное пользователем явное преобразование. Явные преобразования могут встречаться в выражениях приведения типов и подробнее описаны в разделе 6.2.

> **БИЛЛ ВАГНЕР**
>
> В противоположность неявным преобразованиям, явные могут завершаться с ошибками или приводить к потере информации, поскольку они запрашиваются пользователем явно.

Операция преобразования выполняет преобразование из исходного типа, обозначенного типом параметра в операции, в целевой тип, обозначенный типом возвращаемого операцией значения.

Для заданных исходного типа `S` и целевого типа `T`, если `S` и `T` являются обнуляемыми типами, пусть S_0 и T_0 обозначают их базовые типы; в противном случае пусть S_0 и T_0 будут эквивалентны соответственно `S` и `T`. В классе или структуре допускается объявление преобразования из исходного типа `S` в целевой тип `T`, только если выполняются все перечисленные условия:

- S_0 и T_0 — разные типы.
- Либо S_0, либо T_0 является типом класса или структуры, в котором находится объявление операции.
- Ни S_0, ни T_0 не являются *интерфейсом*.
- Не существует преобразований из `S` в `T` или из `T` в `S`, за исключением определенных пользователем.

Во время применения этих правил любые параметры-типы, ассоциированные с `S` или `T`, считаются уникальными типами, не связанными отношением наследования с другими типами, а любые ограничения на эти параметры-типы игнорируются.

Пример:

```
class C<T> {...}

class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...}      // Верно

    public static implicit operator C<string>(D<T> value) {...}   // Верно

   public static implicit operator C<T>(D<T> value) {...}         // Ошибка
}
```

Здесь два первых объявления операций допустимы, поскольку, как указано в разделе 10.10.3, `T` не связан соответственно с `int` и `string`; в обоих случаях эти типы считаются уникальными. Однако третья операция приводит к ошибке, поскольку `C<T>` является базовым классом `D<T>`.

Из второго правила следует, что операция преобразования должна преобразовывать в тип или из типа класса или структуры, в которых объявлена операция.

Например, в типе класса или структуры `C` можно определять преобразования из `C` в `int` и из `int` в `C`, но не из `int` в `bool`.

Невозможно непосредственно переопределить встроенное преобразование. Следовательно, операции преобразования не могут преобразовывать из типа или в тип `object`, поскольку между этим типом и всеми остальными типами уже существуют неявные и явные преобразования. Аналогично, исходным типом преобразования не может быть базовый тип целевого, а целевым типом — базовый тип исходного, поскольку такие преобразования уже существуют.

Однако *возможно* объявлять в обобщенных типах такие операции, которые при определенных аргументах-типах будут определять преобразования, уже существующие в виде встроенных. Пример:

```
struct Convertible<T>
{
   public static implicit operator Convertible<T>(T value) {...}
   public static explicit operator T(Convertible<T> value) {...}
}
```

Здесь, если указать в качестве аргумента-типа `T` класс `object`, вторая операция будет объявлять уже существующее преобразование (существует неявное, а следовательно, также и явное, преобразование из любого типа в тип `object`).

В случаях, когда между двумя типами существует встроенное преобразование, любые определенные пользователем преобразования между этими типами игнорируются. А точнее:

- Если существует встроенное неявное преобразование (раздел 6.1) из типа `S` в тип `T`, все определенные пользователем преобразования (неявные или явные) из `S` в `T` игнорируются.
- Если существует встроенное явное преобразование (раздел 6.2) из типа `S` в тип `T`, все определенные пользователем явные преобразования из `S` в `T` игнорируются. Более того:
  - Если `T` является типом интерфейса, определенные пользователем неявные преобразования из `S` в `T` игнорируются.
  - В противном случае определенные пользователем неявные преобразования из `S` в `T` все еще учитываются.

Для всех типов кроме `object`, операции, объявленные выше в типе `Convertible<T>`, не конфликтуют со встроенными преобразованиями. Пример:

```
void F(int i, Convertible<int> n) {
    i = n;                          // Ошибка
    i = (int)n;                     // Определенное пользователем
                                    // явное преобразование
    n = i;                          // Определенное пользователем
                                    // неявное преобразование
    n = (Convertible<int>)i;        // Определенное пользователем
                                    // неявное преобразование
}
```

Однако для типа `object` определенные пользователем преобразования скрыты встроенными во всех случаях, кроме одного:

```
void F(object o, Convertible<object> n) {
    o = n;                          // Встроенное упаковывающее
                                    // преобразование
    o = (object)n;                  // Встроенное упаковывающее
                                    // преобразование
    n = o;                          // Определенное пользователем
                                    // неявное преобразование
    n = (Convertible<object>)o;     // Встроенное распаковывающее
                                    // преобразование
}
```

Определенные пользователем преобразования не могут преобразовывать из *интерфейсов* и обратно. В частности, это ограничение гарантирует, что при преобразовании в *интерфейс* не будут применяться пользовательские преобразования и что преобразование в *интерфейс* завершится успешно, только если преобразуемый объект действительно реализует указанный *интерфейс*.

Сигнатура операции преобразования состоит из исходного типа и целевого типа (заметьте, что это единственный вид элемента, для которого тип возвращаемого значения входит в сигнатуру). Классификация операции преобразования как `implicit` или `explicit` не относится к сигнатуре. Следовательно, в классе или структуре не могут быть одновременно объявлены операции неявного и явного преобразования, имеющие одинаковые исходные и целевые типы.

Вообще определенные пользователем неявные преобразования следует проектировать таким образом, чтобы они никогда не выбрасывали исключений и никогда не приводили к потере информации. Если определенное пользователем преобразование может генерировать исключения (например, из-за выхода значения исходного аргумента за допустимые пределы) или приводить к потере информации (например, отбрасыванию старших разрядов), это преобразование должно быть объявлено как явное. Пример:

```
using System;

public struct Digit
{
   byte value;

   public Digit(byte value) {
      if (value < 0 || value > 9) throw new ArgumentException();
      this.value = value;
   }

   public static implicit operator byte(Digit d) {
       return d.value;
   }

   public static explicit operator Digit(byte b) {
       return new Digit(b);
   }
}
```

Здесь преобразование из `Digit` в `byte` сделано неявным, поскольку оно никогда не выбрасывает исключений и не приводит к потере информации, однако преобразование из `byte` в `Digit` сделано явным, поскольку `Digit` может представлять лишь подмножество возможных значений `byte`.

10.11. Конструкторы экземпляра

Конструктор экземпляра — это элемент, реализующий действия, необходимые для инициализации экземпляра класса. Конструкторы экземпляра объявляются с помощью *объявлений-конструктора*:

```
объявление-конструктора:
   атрибуты_opt   модификаторы-конструктора_opt   описатель-конструктора
      тело-конструктора

модификаторы-конструктора:
   модификатор-конструктора
   модификаторы-конструктора   модификатор-конструктора

модификатор-конструктора:
   public
   protected
   internal
   private
   extern

описатель-конструктора:
   идентификатор   (   список-формальных-параметров_opt   )
      инициализатор-конструктора_opt

инициализатор-конструктора:
   :   base   (   список-аргументов_opt   )
   :   this   (   список-аргументов_opt   )

тело-конструктора:
   блок
   ;
```

Объявление-конструктора может содержать набор *атрибутов* (раздел 17) и корректную комбинацию четырех модификаторов доступа (раздел 10.3.5) и модификатора `extern`. Один и тот же модификатор не может встречаться в объявлении конструктора несколько раз.

Идентификатор в *описателе-конструктора* должен представлять собой имя класса, в котором объявляется конструктор экземпляра. Если указано любое другое имя, возникает ошибка компиляции.

Необязательный *список-формальных-параметров* конструктора экземпляра подчиняется тем же правилам, что и *список-формальных-параметров* метода (раздел 10.6). Список формальных параметров определяет сигнатуру (раздел 3.6) конструктора экземпляра и влияет на процесс выбора конкретного конструктора для вызова при разрешении перегрузки (раздел 7.5.2).

Каждый из типов, используемых в *списке-формальных-параметров* конструктора экземпляра, должен иметь не более строгий вид доступа, чем сам конструктор (раздел 3.5.4).

Необязательный *инициализатор-конструктора* определяет другой конструктор экземпляра, который будет вызван перед выполнением операторов, расположенных в *теле-конструктора*. Более подробно это описано в разделе 10.11.1.

Когда объявление конструктора содержит модификатор `extern`, конструктор называется **внешним конструктором**. Поскольку внешний конструктор не предоставляет фактической реализации, его *тело-конструктора* состоит из точки с запятой. Для всех остальных конструкторов *тело-конструктора* состоит из *блока*, определяющего операторы, инициализирующие новый экземпляр класса. Он в точности соответствует *блоку* в методе экземпляра, возвращающего `void` (раздел 10.6.10).

Конструкторы экземпляра не наследуются. Следовательно, класс не содержит иных конструкторов, кроме объявленных в нем самом. Если в классе отсутствуют объявления конструкторов экземпляра, автоматически предоставляется конструктор экземпляра по умолчанию (раздел 10.11.4).

Конструкторы экземпляра вызываются с помощью *выражений-создания-объекта* (раздел 7.6.10.1), а также *инициализаторов-конструктора*.

10.11.1. Инициализаторы конструктора

Все конструкторы экземпляра (кроме конструкторов класса `object`) неявно содержат вызов другого конструктора экземпляра непосредственно перед *телом-конструктора*. Этот неявно вызываемый конструктор определяется *инициализатором-конструктора*:

- Инициализатор конструктора экземпляра в форме `base(`*список-аргументов*$_{opt}$`)` приводит к вызову конструктора-экземпляра из непосредственного базового класса. Выбор этого конструктора выполняется с помощью *списка-аргументов* и правил разрешения перегрузки, описанных в разделе 7.5.3. Множество возможных конструкторов экземпляра состоит из всех доступных конструкторов экземпляра, находящихся в непосредственном базовом классе, а если таких нет — из конструктора базового класса по умолчанию (раздел 10.11.4). Если множество пусто или невозможно найти единственный наиболее подходящий конструктор экземпляра, возникает ошибка компиляции.
- Инициализатор конструктора экземпляра в форме `this(`*список-аргументов*$_{opt}$`)` приводит к вызову конструктора экземпляра из самого класса. Выбор конструктора выполняется с помощью *списка-аргументов* и правил разрешения перегрузки, описанных в разделе 7.5.3. Множество возможных конструкторов экземпляра состоит из всех доступных конструкторов экземпляра, объявленных в самом классе. Если множество пусто или невозможно найти единственный наиболее подходящий конструктор, возникает ошибка компиляции. Если объявление конструктора экземпляра содержит инициализатор конструктора, вызывающий этот же конструктор, возникает ошибка компиляции.

ВЛАДИМИР РЕШЕТНИКОВ

Аргумент *инициализатора-конструктора* может иметь тип `dynamic`, только если этот аргумент содержит модификатор `ref` или `out`; в противном случае возникает ошибка компиляции (CS1975). Следовательно, динамическое связывание *инициализатора-конструктора* никогда не выполняется.

Если в конструкторе экземпляра отсутствует инициализатор конструктора, автоматически предоставляется инициализатор в форме `base()`. Таким образом, объявление конструктора экземпляра в форме

```
C(...) {...}
```

в точности эквивалентно объявлению

```
C(...): base() {...}
```

Область видимости параметров, указанных в *списке-формальных-параметров* конструктора экземпляра, включает в себя инициализатор конструктора для данного объявления. Следовательно, инициализатор конструктора может обращаться к параметрам конструктора. Пример:

```
class A
{
    public A(int x, int y) {}
}

class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

Инициализатор конструктора не может получить доступ к создаваемому экземпляру. Следовательно, использование `this` в выражении аргумента в инициализаторе конструктора в данном случае привело бы к ошибке компиляции, равно как и обращение в этом выражении к любому элементу экземпляра с помощью *простого-имени*.

10.11.2. Инициализаторы переменных экземпляра

Когда в конструкторе экземпляра инициализатор конструктора отсутствует или присутствует в форме `base(...)`, этот конструктор неявно выполняет инициализации, определенные *инициализаторами-переменных* в полях экземпляра, объявленных в этом классе. Эти инициализации соответствуют последовательности присваиваний, выполняемых непосредственно при входе в конструктор и перед неявным вызовом конструктора непосредственного базового класса. Инициализаторы переменных выполняются в текстовом порядке, в котором они расположены в объявлении класса.

10.11.3. Выполнение конструктора

Инициализаторы переменных преобразуются в операторы присваивания, которые выполняются перед вызовом конструктора экземпляра базового класса. Такой порядок гарантирует, что все поля экземпляра будут инициализированы своими инициализаторами переменных перед выполнением любых операторов, имеющих доступ к этому экземпляру. Пример:

```
using System;
class A
{
   public A() {
       PrintFields();
   }

   public virtual void PrintFields() {}
}

class B: A
{
   int x = 1;
   int y;

   public B() {
       y = -1;
   }

   public override void PrintFields() {
       Console.WriteLine("x = {0}, y = {1}", x, y);
   }
}
```

Если для создания экземпляра `B` используется `new B()`, вывод будет следующим:

```
x = 1, y = 0
```

`x` имеет значение `1` вследствие того, что инициализатор переменной выполнился раньше вызова конструктора экземпляра базового класса. Однако `y` имеет значение `0` (значение по умолчанию для `int`), так как присваивание значения этой переменной не выполнялось до момента возврата из конструктора базового класса.

Полезно думать об инициализаторах переменных экземпляра и инициализаторах конструкторов как об операторах, которые автоматически вставляются перед *телом-конструктора*. Пример:

```
using System;
using System.Collections;

class A
{
   int x = 1, y = -1, count;

   public A() {
       count = 0;
   }

   public A(int n) {
       count = n;
   }
}

class B: A
{
   double sqrt2 = Math.Sqrt(2.0);
   ArrayList items = new ArrayList(100);
   int max;
```

```
   public B(): this(100) {
       items.Add("элемент по умолчанию");
   }

   public B(int n): base(n – 1) {
       max = n;
   }
}
```

В этом примере присутствуют несколько инициализаторов переменных; здесь также используются инициализаторы конструктора в обеих формах (`base` и `this`). Пример соответствует приведенному ниже коду, в котором каждый комментарий обозначает автоматически добавленный оператор (синтаксис, использованный для автоматически добавленных вызовов конструктора, некорректен и используется просто для демонстрации механизма).

```
using System.Collections;


class A
{
   int x, y, count;

   public A(int n) {
      x = 1;                        // Инициализатор переменной
      y = -1;                       // Инициализатор переменной
      object();                     // Вызов конструктора object()
      count = n;
   }

   public A() {
      x = 1;                        // Инициализатор переменной
      y = -1;                       // Инициализатор переменной
      object();                     // Вызов конструктора object()
      count = 0;
   }
}

class B: A
{
   double sqrt2;
   ArrayList items;
   int max;

   public B(): this(100) {
      B(100);                       // Вызов конструктора B(int)
      items.Add("default");
   }

   public B(int n): base(n – 1) {
      sqrt2 = Math.Sqrt(2.0);       // Инициализатор переменной
      items = new ArrayList(100);   // Инициализатор переменной
      A(n – 1);                     // Вызов конструктора A(int)
      max = n;
   }
}
```

10.11.4. Конструкторы по умолчанию

Если в классе отсутствуют объявления конструкторов экземпляра, автоматически добавляется конструктор экземпляра по умолчанию. Он просто вызывает конструктор без параметров непосредственного базового класса. Если базовый класс не содержит доступного конструктора экземпляра без параметров, возникает ошибка компиляции. Если класс является абстрактным, конструктор по умолчанию имеет вид доступа `protected`. В противном случае он имеет вид доступа `public`. Следовательно, конструктор по умолчанию всегда представлен в форме

```
protected C(): base() {}
```

или

```
public C(): base() {}
```

где `C` — имя класса.

Пример:

```
class Message
{
    object sender;
    string text;
}
```

Здесь добавляется конструктор по умолчанию, так как класс не содержит объявлений конструкторов экземпляра. Следовательно, данный пример в точности эквивалентен следующему:

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

10.11.5. Закрытые конструкторы

Если в классе `T` присутствуют только конструкторы экземпляра, объявленные как `private`, классы, расположенные за пределами объявления `T`, не могут наследовать от него или непосредственно создавать его экземпляры. Следовательно, если класс содержит только статические элементы и не предназначен для создания экземпляров, их создание можно запретить, добавив в класс пустой закрытый конструктор экземпляра. Пример:

```
public class Trig
{
   private Trig() {}        // Предотвращение создания экземпляров

   public const double PI = 3.14159265358979323846;

   public static double Sin(double x) {...}
   public static double Cos(double x) {...}
   public static double Tan(double x) {...}
}
```

Класс `Trig` объединяет родственные методы и константы, однако не предназначен для создания экземпляров. Поэтому он объявляет единственный пустой закрытый конструктор экземпляра. Чтобы предотвратить автоматическую генерацию конструктора по умолчанию, в классе должен быть объявлен хотя бы один конструктор экземпляра.

> **БИЛЛ ВАГНЕР**
>
> Если класс не предназначен для создания экземпляров, лучше сделать его статическим. Однако если вы реализуете шаблон «Одиночка» (Singleton), закрытый конструктор гарантирует, что единственный объект может быть создан только вашим методом-фабрикой.

> **ДЖОЗЕФ АЛЬБАХАРИ**
>
> Если вашей целью является предотвращение создания экземпляров, проще объявить класс как `static`.
> У закрытых конструкторов также есть и более хитроумное предназначение, основанное на том факте, что закрытый конструктор все равно может быть вызван из статических элементов того же класса. Иногда предоставление лишь статических методов в качестве единственного открытого средства создания экземпляра класса дает несколько преимуществ. Например, в случае неизменяемых объектов данный подход можно использовать для реализации прозрачной системы кэширования объектов.

10.11.6. Необязательные параметры конструкторов экземпляра

Инициализаторы конструкторов в форме `this(...)` обычно используются вместе с перегрузкой для реализации необязательных параметров конструкторов экземпляра. Пример:

```
class Text
{
   public Text(): this(0, 0, null) {}

   public Text(int x, int y): this(x, y, null) {}

   public Text(int x, int y, string s) {
       // Фактическая реализация конструктора
   }
}
```

В этом примере два первых экземпляра конструктора просто предоставляют значения по умолчанию для отсутствующих аргументов. Они используют инициализатор конструктора `this(...)` для вызова третьего конструктора экземпляра, который фактически выполняет действия по инициализации нового экземпляра. Результатом этого является возможность использования необязательных параметров конструктора:

```
Text t1 = new Text();                    // То же, что Text(0, 0, null)
Text t2 = new Text(5, 10);               // То же, что Text(5, 10, null)
Text t3 = new Text(5, 20, "Привет");
```

10.12. Статические конструкторы

Статический конструктор — это элемент, реализующий действия, необходимые для инициализации закрытого класса. Статические конструкторы объявляются с помощью *объявлений-статических-конструкторов*:

```
объявление-статического-конструктора:
   атрибуты_opt   модификаторы-статического-конструктора   идентификатор
      (   )   тело-статического-конструктора

модификаторы-статического-конструктора:
   extern_opt   static
   static   extern_opt

тело-статического-конструктора:
   блок
   ;
```

Объявление-статического-конструктора может содержать набор *атрибутов* (раздел 17) и модификатор `extern` (раздел 10.6.7).

Идентификатор в *объявлении-статического-конструктора* должен носить имя класса, в котором объявляется конструктор. Если указано другое имя, возникает ошибка компиляции.

Когда объявление статического конструктора содержит модификатор `extern`, статический конструктор называется **внешним статическим конструктором**. Поскольку объявление внешнего статического конструктора не предоставляет фактической реализации, *тело-статического-конструктора* в нем состоит из точки с запятой. Для всех остальных объявлений статических конструкторов *тело-статического-конструктора* состоит из *блока*, который определяет операторы, выполняющиеся для инициализации класса. Оно полностью соответствует *телу-метода* для статического метода, возвращающего `void` (раздел 10.6.10).

Статические конструкторы не наследуются и не могут вызываться явно.

Статический конструктор для закрытого класса выполняется не более одного раза в конкретном домене приложения. Выполнение статического конструктора начинается при возникновении в домене приложения одного из следующих событий:

- Создается экземпляр класса.
- Происходит обращение к любому статическому элементу класса.

ДЖОН СКИТ

Очень вероятно, что наличие пустого статического конструктора повлияет на поведение кода, так как оно может изменить момент, в который начнут выполняться инициализаторы статических полей. Если вы решите воспользоваться преимуществами такого

> подхода, я настоятельно рекомендую, чтобы вы хотя бы добавили в статический конструктор комментарий, поясняющий, почему и как вы полагаетесь на его наличие, чтобы избежать его возможного удаления излишне активным программистом, сопровождающим ваш код.

Если класс содержит метод `Main` (раздел 3.1), с которого начинается выполнение программы, статический конструктор этого класса выполняется до вызова `Main`.

Для инициализации нового закрытого класса сначала создается новый набор статических полей (раздел 10.5.1) для этого конкретного типа. Каждое из статических полей инициализируется своим значением по умолчанию (раздел 5.2). Затем для них выполняются инициализаторы статических полей (раздел 10.4.5.1). И наконец, выполняется статический конструктор. Пример:

```
using System;

class Test
{
   static void Main()
   {
      A.F();
      B.F();
   }
}

class A
{
   static A()
   {
       Console.WriteLine("Инициализация A");
   }
   public static void F()
   {
       Console.WriteLine("A.F");
   }
}

class B
{
   static B()
   {
       Console.WriteLine("Инициализация B");
   }
   public static void F()
   {
       Console.WriteLine("B.F");
   }
}
```

Приведенный код должен вывести следующее:

```
Инициализация A
A.F
Инициализация B
B.F
```

Такой вывод получается вследствие того, что вызов `A.F` приводит к выполнению статического конструктора класса `A`, а вызов `B.F` — к выполнению статического конструктора класса `B`.

Существует возможность создать циклические зависимости, позволяющие наблюдать состояние, в котором статические поля с инициализаторами переменных будут иметь значения по умолчанию.

Пример:

```
using System;

class A
{
   public static int X;

   static A()
   {
       X = B.Y + 1;
   }
}

class B
{
   public static int Y = A.X + 1;

   static B() { }

   static void Main()
   {
       Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
   }
}
```

В данном случае вывод будет следующим:

```
X = 1, Y = 2
```

Для выполнения метода `Main` система сначала выполняет инициализатор для `B.Y`, перед тем как будет вызван статический конструктор класса `B`. Инициализатор `Y` приводит к выполнению статического конструктора класса `A`, так как использует значение `A.X`. В свою очередь, статический конструктор `A` переходит к вычислению значения `X` и в процессе вычисления получает значение по умолчанию для `Y`, равное нулю. Поэтому `A.X` инициализируется единицей. Затем процесс выполнения инициализаторов статических полей и статического конструктора в классе `A` завершается, и продолжается вычисление первоначального значения `Y`, которое становится равным `2`.

Поскольку для каждого закрытого сконструированного класса статический конструктор выполняется ровно один раз, он представляет собой удобное место для осуществления проверок времени выполнения параметра-типа, который невозможно проверить с помощью ограничений (раздел 10.1.5) на этапе компиляции. Например, следующий тип использует статический конструктор, чтобы удостовериться, что аргумент-тип является перечислением:

```
class Gen<T> where T : struct
{
```

```
   static Gen()
   {
      if (!typeof(T).IsEnum)
      {
         throw new ArgumentException("T должен быть перечислением");
      }
   }
}
```

10.13. Деструкторы

Деструктор — это элемент, реализующий действия, необходимые для уничтожения экземпляра класса. Деструктор объявляется с помощью *объявления-деструктора*:

```
объявление-деструктора:
   атрибуты_opt   extern_opt   ~   идентификатор   (   )   тело-деструктора

тело-деструктора:
   блок
   ;
```

Объявление-деструктора может содержать набор *атрибутов* (раздел 10.17).

Идентификатор в *объявлении-деструктора* должен носить имя класса, в котором объявляется деструктор. Если указано другое имя, возникает ошибка компиляции.

Если объявление деструктора содержит модификатор `extern`, деструктор называется **внешним деструктором**. Поскольку объявление внешнего деструктора не предоставляет фактической реализации, *тело-деструктора* в нем состоит из точки с запятой. Для всех остальных деструкторов *тело-деструктора* состоит из *блока*, определяющего операторы, выполняемые для уничтожения экземпляра класса. *Тело-деструктора* в точности соответствует *телу-метода* в методе экземпляра, возвращающем `void` (раздел 10.6.10).

Деструкторы не наследуются. Следовательно, класс не содержит иных деструкторов, кроме того, который может быть объявлен в этом классе.

Поскольку в деструкторе не допускаются параметры, его нельзя перегрузить, а это значит, что в классе может быть не более одного деструктора.

Деструкторы вызываются автоматически и не могут быть вызваны явно. Экземпляр становится пригодным для уничтожения, если он больше не может использоваться никаким кодом. Выполнение деструктора экземпляра может произойти в любой момент, после того как экземпляр станет пригодным для уничтожения. Когда экземпляр уничтожается, вызываются деструкторы в цепочке наследования этого экземпляра, начиная от деструктора самого нижнего уровня и заканчивая деструктором самого верхнего. Деструктор может выполниться в любом потоке. Более подробные правила, определяющие, когда и как выполняется деструктор, приведены в разделе 3.9.

ЭРИК ЛИППЕРТ

Очевидно, что код, находящийся в деструкторе, потенциально выполняется в окружении, сильно отличающемся от того, в котором выполняется любой другой код вашей программы. Поэтому выполнять здесь какие-то сложные, опасные, приводящие к побочным эффектам или длительные действия — очень плохая идея. В частности, следующий код, порождающий сложный побочный эффект в виде вывода на консоль, предназначен для учебных целей и *не является* примером кода, который вы должны писать в настоящем деструкторе.

Пример:

```
using System;
class A
{
   ~A()
   {
       Console.WriteLine("Деструктор A ");
   }
}

class B : A
{
   ~B()
   {
       Console.WriteLine("Деструктор B");
   }
}

class Test
{
   static void Main()
   {
      B b = new B();
      b = null;
      GC.Collect();
      GC.WaitForPendingFinalizers();
   }
}
```

Этот код выводит следующее:

```
Деструктор B
Деструктор A
```

Вывод обусловлен тем, что деструкторы в цепочке наследования выполняются в порядке от самого нижнего до самого верхнего.

ЭРИК ЛИППЕРТ

Обычно термином «деструктор» обозначают детерминированный метод очистки, а термином «финализатор» — недетерминированный метод очистки, вызываемый сборщиком мусора. «Деструктор» — пример неудачного названия в C#; в идеале эти методы должны были бы называться «финализаторами», а метод `Dispose` из интерфейса `IDisposable` — «деструктором».

Деструкторы реализуются путем переопределения виртуального метода `Finalize` из класса `System.Object`. В программах на C# не допускается переопределять данный метод или непосредственно вызывать его (а также его переопределения). Рассмотрим программу:

```
class A
{
   override protected void Finalize() { }        // Ошибка

   public void F()
   {
       this.Finalize();                          // Ошибка
   }
}
```

Этот код содержит две ошибки.

Компилятор ведет себя так, как если бы этого метода и его переопределений вообще не существовало. Следовательно, программа

```
class A
{
    void Finalize() { }                            // Допустимо
}
```

является корректной, а метод `Finalize` из `System.Object` скрывается объявленным в ней методом.

Поведение при выбрасывании исключения в деструкторе описано в разделе 16.3.

10.14. Итераторы

Функциональный элемент (раздел 7.5), реализованный с помощью блока итератора, называется **итератором**.

Блок итератора может использоваться в качестве тела функционального элемента при условии, что тип возвращаемого этим элементом значения относится к одному из интерфейсов-перечислителей (раздел 10.14.1) или к одному из перечислимых интерфейсов (раздел 10.14.2). Этот блок может располагаться в *теле-метода*, *теле-операции* или *теле-кода-доступа*, а события, конструкторы экземпляра, статические конструкторы и деструкторы не могут быть реализованы как итераторы.

Если функциональный элемент реализуется с использованием блока итератора, указание параметров `ref` или `out` в списке формальных параметров этого элемента приведет к ошибке компиляции.

10.14.1. Интерфейсы-перечислители

К **интерфейсам-перечислителям** относятся необобщенный интерфейс `System.Collections.IEnumerator` и все специализации обобщенного интерфейса `System.`

`Collections.Generic.IEnumerator<T>`. Для краткости в этой главе будут использоваться соответственно названия `IEnumerator` и `IEnumerator<T>`.

> **ДЖОН СКИТ**
>
> Я бы хотел, чтобы вместо терминов «перечислимый» и «перечислитель» использовались «итерируемый» и «итератор», а для упомянутых интерфейсов — имена `IIterable<T>` и `IIterator<T>`. Конечно, две буквы `I` подряд выглядят ужасно, но такие имена внесли бы большее лексическое различие между итераторами и перечислителями. Формулировка «перечислять перечисление» излишне многословна.

10.14.2. Перечислимые интерфейсы

К **перечислимым интерфейсам** относятся необобщенный интерфейс `System.Collections.IEnumerable` и все специализации обобщенного интерфейса `System.Collections.Generic.IEnumerable<T>`. Для краткости в этой главе будут использоваться соответственно `IEnumerable` и `IEnumerable<T>`.

10.14.3. Результирующий тип

Итератор создает последовательность значений одного типа. Этот тип называется **результирующим типом** итератора.

- Результирующим типом итератора, возвращающего `IEnumerator` или `IEnumerable`, является `object`.
- Результирующим типом итератора, возвращающего `IEnumerator<T>` или `IEnumerable<T>`, является `T`.

10.14.4. Объекты-перечислители

Когда функциональный элемент, возвращающий тип интерфейса-перечислителя, реализуется с помощью блока итератора, вызов этого элемента не приводит к немедленному выполнению кода в блоке. Вместо этого создается и возвращается **объект-перечислитель**. Этот объект содержит код, указанный в блоке итератора, и выполнение данного кода происходит только при вызове метода `MoveNext` для объекта-перечислителя. Объект-перечислитель обладает следующими характеристиками:

- Он реализует интерфейсы `IEnumerator` и `IEnumerator<T>`, где `T` — результирующий тип итератора.
- Он реализует `System.IDisposable`.
- Он инициализируется копиями значений аргументов (если они есть) и значением экземпляра, переданными в функциональный элемент.
- У него есть четыре возможных состояния: **до, выполняется, приостановлен** и **после** — и первоначально он находится в состоянии **до**.

> **ДЖОН СКИТ**
>
> То, что *никакой* код из блока итератора не выполняется до первого вызова `MoveNext()`, раздражает. Это означает, что если вам требуется проверить какие-либо параметры-значения, вам придется фактически использовать два метода: обычный метод, который выполняет соответствующую проверку и вызывает второй метод, который реализуется с помощью блока итератора. В идеальном варианте было бы неплохо иметь некую конструкцию для обозначения секции блока итератора, выполняемой перед созданием конечного автомата.

Объект-перечислитель обычно представляет собой экземпляр класса-перечислителя, генерируемого компилятором, который содержит код из блока итератора и реализует интерфейсы-перечислители, но возможны и другие варианты его реализации. Если класс-перечислитель генерируется компилятором, он будет непосредственно или косвенно вложен в класс, содержащий функциональный элемент, иметь вид доступа `private` и имя, зарезервированное для использования компилятором (раздел 2.4.2).

Объект-перечислитель может реализовывать и другие интерфейсы помимо указанных выше.

В следующих разделах описано точное поведение элементов `MoveNext`, `Current` и `Dispose` в реализациях интерфейсов `IEnumerable` и `IEnumerable<T>`, предоставляемых объектом-перечислителем.

Заметьте, что объекты-перечислители не поддерживают метод `IEnumerator.Reset`. Вызов этого метода приведет к выбрасыванию исключения `System.NotSupportedException`.

10.14.4.1. Метод MoveNext

Метод `MoveNext` в объекте-перечислителе содержит код из блока итератора. Вызов метода приводит к выполнению этого кода и установке подходящего значения свойства `Current` в этом объекте. Точные действия, выполняемые методом `MoveNext`, зависят от состояния объекта-перечислителя на момент вызова метода:

- Если объект-перечислитель находится в состоянии **до**, вызов `MoveNext` делает следующее:
  - Изменяет состояние на **выполняется**.
  - Инициализирует параметры (включая `this`) блока итератора значениями аргументов и значением экземпляра, сохраненными при инициализации объекта-перечислителя.
  - Выполняет блок итератора с самого начала, пока выполнение не будет прервано (как описано ниже).
- Если объект-перечислитель находится в состоянии **выполняется**, результат вызова `MoveNext` не определен.
- Если объект-перечислитель находится в состоянии **приостановлен**, вызов `MoveNext` делает следующее:

  - Изменяет состояние на **выполняется**.
  - Возвращает значениям всех локальных переменных и параметров (включая `this`) значения, сохраненные во время последней приостановки выполнения блока итератора. Заметьте, что содержимое любых объектов, на которые ссылаются эти переменные, могло измениться с момента предыдущего вызова `MoveNext`.
  - Возобновляет выполнение блока итератора с позиции, непосредственно следующей за оператором `yield return`, ставшем причиной приостановки, и продолжает выполнение, пока оно не будет прервано (как описано ниже).
- Если объект-перечислитель находится в состоянии **после**, вызов `MoveNext` возвращает `false`.

Когда `MoveNext` выполняет блок итератора, выполнение может быть прервано четырьмя способами: оператором `yield return`, оператором `yield break`, достижением конца блока итератора или исключением, вышедшим за пределы блока итератора.

- Когда встречается оператор `yield return` (раздел 8.14):
  - Выражение, указанное в операторе, вычисляется, неявно преобразуется в результирующий тип и присваивается свойству `Current` объекта-перечислителя.
  - Выполнение тела итератора приостанавливается. Происходит сохранение значений всех локальных переменных и параметров (включая `this`), а также позиции оператора `yield return`. Если оператор `yield return` находится внутри одного или нескольких блоков `try`, соответствующие им блоки `finally` в этот момент *не* выполняются.
  - Состояние объекта-перечислителя изменяется на **приостановлен**.
  - Метод `MoveNext` возвращает `true`, сигнализируя, что произошел успешный переход к следующему значению.
- Когда встречается оператор `yield break` (раздел 8.14):
  - Если оператор находится внутри одного или нескольких блоков `try`, выполняются соответствующие им блоки `finally`.
  - Состояние объекта-перечислителя изменяется на **после**.
  - Метод `MoveNext` возвращает `false`, сигнализируя, что итерирование завершено.
- Когда достигнут конец тела итератора:
  - Состояние объекта-перечислителя изменяется на **после**.
  - Метод `MoveNext` возвращает `false`, сигнализируя, что итерирование завершено.
- Когда выбрасывается исключение, вышедшее за пределы блока итератора:
  - В процессе распространения исключения будут выполнены соответствующие блоки `finally` в теле итератора.

- Состояние объекта-перечислителя изменяется на **после**.
- Исключение передается коду, вызвавшему метод `MoveNext`.

10.14.4.2. Свойство Current

На значение свойства `Current` в объекте-перечислителе влияют операторы `yield return` в блоке итератора.

Когда объект-перечислитель находится в состоянии **приостановлен**, значением `Current` является значение, установленное предыдущим вызовом `MoveNext`. Когда объект-перечислитель находится в состоянии **до**, **выполняется** или **после**, результат доступа к `Current` не определен.

Для итератора, результирующий тип которого отличен от `object`, результат доступа к `Current` через реализацию `IEnumerable` объектом-перечислителем соответствует доступу к `Current` через реализацию `IEnumerator<T>` объектом-перечислителем и последующим приведением результата к типу `object`.

10.14.4.3. Метод Dispose

Метод `Dispose` используется для завершения процесса итерации путем перевода объекта-перечислителя в состояние **после**.

- Если объект-перечислитель находится в состоянии **до**, вызов метода `Dispose` изменяет состояние на **после**.
- Если объект-перечислитель находится в состоянии **выполняется**, результат вызова `Dispose` не определен.
- Если объект-перечислитель находится в состоянии **приостановлен**, вызов `Dispose` делает следующее:
  - Изменяет состояние на **выполняется**.
  - Выполняет все блоки `finally`, как если бы последний выполненный оператор `yield return` был оператором `yield break`. Если данное действие приводит к вызову исключения и передаче его за пределы тела итератора, состояние объекта-перечислителя изменяется на **после** и исключение передается в код, вызвавший метод `Dispose`.
  - Изменяет состояние на **после**.
- Если объект-перечислитель находится в состоянии **после**, вызов `Dispose` ни на что не влияет.

10.14.5. Перечислимые объекты

Если функциональный элемент, возвращающий тип перечислимого интерфейса, реализован с помощью блока итератора, вызов этого элемента не приводит к немедленному выполнению кода в блоке итератора. Вместо этого создается и возвращается **перечислимый объект**. Метод `GetEnumerator` этого объекта возвращает объект-перечислитель, содержащий код, указанный в блоке итератора, и выпол-

нение данного кода происходит только при вызове метода `MoveNext` для объекта-перечислителя. Перечислимый объект обладает следующими характеристиками:

- Он реализует интерфейсы `IEnumerable` и `IEnumerable<T>`, где `T` — результирующий тип итератора.
- Он инициализируется копиями значений аргументов (если они есть) и значением экземпляра, переданными в функциональный элемент.

Перечислимый объект обычно представляет собой экземпляр перечислимого класса, генерируемого компилятором, который содержит код из блока итератора и реализует перечислимые интерфейсы, но возможны и другие варианты его реализации. Если перечислимый класс генерируется компилятором, он будет непосредственно или косвенно вложен в класс, содержащий функциональный элемент, иметь вид доступа `private` и имя, зарезервированное для использования компилятором (раздел 2.4.2).

Перечислимый объект может реализовывать и другие интерфейсы, помимо указанных выше. В частности, перечислимый объект также реализует `IEnumerator` и `IEnumerator<T>`, в результате чего он может использоваться одновременно как в качестве перечислимого объекта, так и в качестве перечислителя. Если он реализуется таким образом, при первом вызове метода `GetEnumerator` перечислимого объекта возвращается сам перечислимый объект. Последующие вызовы `GetEnumerator`, если они есть, возвращают копию перечислимого объекта. Таким образом, у каждого возвращаемого перечислителя будет существовать свое собственное состояние, а изменения в одном перечислителе не будут влиять на другой.

10.14.5.1. Метод GetEnumerator

Перечислимый объект предоставляет реализации методов `GetEnumerator` из интерфейсов `IEnumerable` и `IEnumerable<T>`. Два метода `GetEnumerator` разделяют общую реализацию, которая получает и возвращает доступный объект-перечислитель. Этот объект инициализируется значениями аргументов и значением экземпляра, сохраненными при инициализации перечислимого объекта, но в том, что касается всего остального, объект-перечислитель функционирует, как описано в разделе 10.14.4.

10.14.6. Пример реализации

В этом разделе описана возможная реализация итераторов с помощью стандартных конструкций C#. Представленная реализация основна на принципах, используемых компилятором Microsoft C#, но ни в коем случае не является обязательной или единственно возможной.

> **БИЛЛ ВАГНЕР**
>
> Изучая этот пример, заметьте, что компилятор просто создает весь код, который вы бы написали, если бы создавали свой собственный вложенный класс-перечислитель.

> Использование итератора помогает избежать огромного количества повторяющейся работы и повышает читабельность вашего кода.

Приведенный ниже класс `Stack<T>` реализует метод `GetEnumerator` с помощью итератора. Итератор перечисляет элементы стека в порядке от вершины ко дну.

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T> : IEnumerable<T>
{
   T[] items;
   int count;

   public void Push(T item)
   {
      if (items == null)
      {
              items = new T[4];
      }
      else if (items.Length == count)
      {
         T[] newItems = new T[count * 2];
         Array.Copy(items, 0, newItems, 0, count);
         items = newItems;
      }
      items[count++] = item;
   }

   public T Pop()
   {
      T result = items[--count];
      items[count] = default(T);
      return result;
   }

   public IEnumerator<T> GetEnumerator()
   {
       for (int i = count - 1; i >= 0; --i) yield return items[i];
   }
}
```

Метод `GetEnumerated` может быть преобразован в экземпляр сгенерированного компилятором класса-перечислителя, содержащего код из блока итератора, как показано в следующем примере:

```
class Stack<T>: IEnumerable<T>
{
   ...
   public IEnumerator<T> GetEnumerator() {
       return new __Enumerator1(this);
   }
```

продолжение ⇗

```
   class __Enumerator1: IEnumerator<T>, IEnumerator
   {
      int __state;
      T __current;
      Stack<T> __this;
      int i;

      public __Enumerator1(Stack<T> __this) {
              this.__this = __this;
      }

      public T Current {
              get { return __current; }
       }

      object IEnumerator.Current {
              get { return __current; }
      }

      public bool MoveNext() {
         switch (__state) {
            case 1: goto __state1;
            case 2: goto __state2;
         }
         i = __this.count - 1;
        __loop:
         if (i < 0) goto __state2;
         __current = __this.items[i];
         __state = 1;
         return true;
        __state1:
         --i;
         goto __loop;
        __state2:
         __state = 2;
         return false;
      }

      public void Dispose() {
              __state = 2;
      }

      void IEnumerator.Reset() {
              throw new NotSupportedException();
      }
   }
}
```

В приведенном варианте преобразования код в блоке итератора превращается в конечный автомат и размещается в методе `MoveNext` в классе-перечислителе. Кроме того, локальная переменная `i` превращается в поле в объекте-перечислителе, поэтому она продолжает существовать между вызовами `MoveNext`.

Следующий пример выводит простую таблицу умножения для чисел от `1` до `10`. Метод `FromTo` в примере возвращает перечислимый объект и реализован с помощью итератора.

```
using System;
using System.Collections.Generic;

class Test
{
   static IEnumerable<int> FromTo(int from, int to)
   {
       while (from <= to) yield return from++;
   }

   static void Main()
   {
      IEnumerable<int> e = FromTo(1, 10);
      foreach (int x in e)
      {
         foreach (int y in e)
         {
             Console.Write("{0,3} ", x * y);
         }
         Console.WriteLine();
      }
   }
}
```

Метод `FromTo` может быть преобразован в экземпляр сгенерированного компилятором перечислимого класса, содержащего код из блока итератора, как показано в следующем примере:

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
   ...
   static IEnumerable<int> FromTo(int from, int to) {
       return new __Enumerable1(from, to);
   }

   class __Enumerable1:
      IEnumerable<int>, IEnumerable,
      IEnumerator<int>, IEnumerator
   {
      int __state;
      int __current;
      int __from;
      int from;
      int to;
      int i;

      public __Enumerable1(int __from, int to) {
         this.__from = __from;
         this.to = to;
      }
```

продолжение ⇨

```
      public IEnumerator<int> GetEnumerator() {
         __Enumerable1 result = this;
         if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
            result = new __Enumerable1(__from, to);
            result.__state = 1;
         }
         result.from = result.__from;
         return result;
      }

      IEnumerator IEnumerable.GetEnumerator() {
              return (IEnumerator)GetEnumerator();
      }

      public int Current {
              get { return __current; }
      }

      object IEnumerator.Current {
              get { return __current; }
      }

      public bool MoveNext() {
         switch (__state) {
            case 1:
            if (from > to) goto case 2;
            __current = from++;
            __state = 1;
            return true;
            case 2:
            __state = 2;
            return false;
            default:
            throw new InvalidOperationException();
         }
      }

      public void Dispose() {
              __state = 2;
      }

      void IEnumerator.Reset() {
         throw new NotSupportedException();
      }
    }
}
```

Перечислимый класс реализует как перечислимые интерфейсы, так и интерфейсы-перечислители, что позволяет использовать его одновременно и в качестве перечислимого объекта, и в качестве перечислителя. При первом вызове метода `GetEnumerator` возвращается сам перечислимый объект. Последующие вызовы `GetEnumerator`, если они есть, возвращают копию перечислимого объекта. Таким образом, каждый возвращаемый перечислитель обладает своим собственным состоянием, и изменения в одном перечислителе не затронут другой. Для гарантии потокобезопасности используется метод `Interlocked.CompareExchange`.

Параметры `from` и `to` превращаются в поля в перечислимом классе. Поскольку `from` изменяется в блоке итератора, вводится дополнительное поле `__from`, хранящее первоначальное значение, заданное для `from` в каждом перечислителе.

Метод `MoveNext` выбрасывает исключение `InvalidOperationException`, если вызывается, когда значение `__state` равно `0`. Это защищает от использования перечислимого объекта в качестве объекта-перечислителя без предварительного вызова `GetEnumerator`.

Следующий пример демонстрирует простой класс, реализующий дерево. Класс `Tree<T>` реализует свой метод `GetEnumerator` с помощью итератора. Последний перечисляет все элементы дерева в инфиксном порядке.

```
using System;
using System.Collections.Generic;

class Tree<T> : IEnumerable<T>
{
   T value;
   Tree<T> left;
   Tree<T> right;

   public Tree(T value, Tree<T> left, Tree<T> right)
   {
      this.value = value;
      this.left = left;
      this.right = right;
   }

   public IEnumerator<T> GetEnumerator() {
      if (left != null) foreach (T x in left) yield x;
      yield value;
      if (right != null) foreach (T x in right) yield x;
   }
}

class Program
{
   static Tree<T> MakeTree<T>(T[] items, int left, int right)
   {
      if (left > right) return null;
      int i = (left + right) / 2;
      return new Tree<T>(items[i],
         MakeTree(items, left, i - 1),
         MakeTree(items, i + 1, right));
   }

   static Tree<T> MakeTree<T>(params T[] items)
   {
       return MakeTree(items, 0, items.Length - 1);
   }

   // Программа выводит:
   // 1 2 3 4 5 6 7 8 9
   // Пн Вт Ср Чт Пт Сб Вс
```

продолжение ☞

```
   static void Main()
   {
      Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
      foreach (int i in ints) Console.Write("{0} ", i);
      Console.WriteLine();

      Tree<string> strings = MakeTree(
              "Пн", "Вт", "Ср", "Чт", "Пт", "Сб", "Вс");
      foreach (string s in strings) Console.Write("{0} ", s);
      Console.WriteLine();
   }
}
```

Метод `GetEnumerator` может быть преобразован в экземпляр сгенерированного компилятором класса-перечислителя, содержащего код из блока итератора, как показано в следующем примере:

```
class Tree<T>: IEnumerable<T>
{
   ...
   public IEnumerator<T> GetEnumerator() {
       return new __Enumerator1(this);
   }

   class __Enumerator1 : IEnumerator<T>, IEnumerator
   {
      Node<T> __this;
      IEnumerator<T> __left, __right;
      int __state;
      T __current;

      public __Enumerator1(Node<T> __this) {
              this.__this = __this;
      }

      public T Current {
              get { return __current; }
      }

      object IEnumerator.Current {
              get { return __current; }
      }

      public bool MoveNext() {
         try {
            switch (__state) {

            case 0:
               __state = -1;
               if (__this.left == null)
                   goto __yield_value;
               __left = __this.left.GetEnumerator();
               goto case 1;

            case 1:
               __state = -2;
               if (!__left.MoveNext())
                   goto __left_dispose;
```

```
            __current = __left.Current;
            __state = 1;
            return true;

             __left_dispose:
            __state = -1;
            __left.Dispose();

             __yield_value:
            __current = __this.value;
            __state = 2;
            return true;

         case 2:
            __state = -1;
            if (__this.right == null) goto __end;
            __right = __this.right.GetEnumerator();
            goto case 3;
         case 3:
            __state = -3;
            if (!__right.MoveNext()) goto __right_dispose;
            __current = __right.Current;
            __state = 3;
            return true;

             __right_dispose:
            __state = -1;
            __right.Dispose();

             __end:
            __state = 4;
            break;
          }
           }

      finally {
              if (__state < 0) Dispose();
      }
      return false;
   }

   public void Dispose() {
      try {
         switch (__state) {

            case 1:
            case -2:
                    __left.Dispose();
                    break;

            case 3:
            case -3:
                    __right.Dispose();
                    break;
         }
      }
```

продолжение ⇗

```
            finally {
                    __state = 4;
            }
        }

        void IEnumerator.Reset() {
                throw new NotSupportedException();
        }
    }
}
```

Временные данные, сгенерированные компилятором для операторов `foreach`, помещаются в поля `__left` и `__right` в объекте-перечислителе. Поле `__state` этого объекта обновляется таким образом, чтобы при выбрасывании исключения осуществился корректный вызов метода `Dispose()`. Заметьте, что невозможно написать преобразованный код с помощью простых операторов `foreach`.

Глава 11
Структуры

Структуры схожи с классами в том, что представляют собой структуры данных с элементами-данными и элементами-функциями. Однако в отличие от классов структуры являются типом-значением и не требуют размещения в куче. Переменные типа `struct` хранят непосредственно данные, в то время как переменные типа класса хранят ссылку на данные, которая называется объектом.

> **ЭРИК ЛИППЕРТ**
>
> Утверждение «структуры не требуют размещения в куче» не эквивалентно утверждению «все экземпляры всех структур всегда размещаются в стеке». Во-первых, последнее утверждение некорректно: память для поля `DateTime` в классе `Customer` будет выделена в куче вместе с остальной памятью, используемой этим классом. Во-вторых, будет ли локальная переменная типа-значения размещена путем изменения регистра стека в процессоре, зависит от реализации конкретной версии платформы .NET. В спецификации указана возможность оптимизации для данного случая, но не указано, что требуется определенный способ размещения.

Структуры очень полезны для небольших структур данных, которые имеют значимую семантику. Комплексные числа, точки в системе координат или пары слово–значение в словаре — все это хорошие примеры структур. Их особенность заключается в том, что они содержат небольшое число элементов-данных, не требуют использования наследования или проверки на равенство ссылок и могут быть легко реализованы с использованием значимой семантики, в которой операция присваивания приводит к копированию значений, а не ссылок.

Как описано в разделе 4.1.4, простые типы C#, такие как `int`, `double` и `bool`, фактически являются структурными типами. Поэтому в C# можно использовать структуры и перегрузку операций для реализации новых «простейших» типов. Два примера таких типов приведены в конце этой главы (раздел 11.4).

> **ДЖОН СКИТ**
>
> Хотя я всегда ценил *возможность* создания пользовательских типов-значений, мне никогда это не требовалось. В некоторых случаях, однако, такая возможность может оказаться чрезвычайно полезной. Например, в данный момент я занимаюсь портированием API для работы с датами и временем из языка Java, и у нас есть три разных типа-значения, в основе которых лежит `long`. В Java для эффективности эти типы часто
>
> *продолжение* ✍

представлены как обычные значения типа `long`, однако использование различных типов с различными операциями сделало код на C# *значительно* более читабельным.

БИЛЛ ВАГНЕР

Все простые типы являются неизменяемыми. Любые созданные вами структуры также должны быть неизменяемы.

11.1. Объявления структур

Объявление-структуры это *объявление-типа* (раздел 9.6), которое объявляет новую структуру:

объявление-структуры:
  атрибуты$_{opt}$ *модификаторы*$_{opt}$ **partial**$_{opt}$ **struct** *идентификатор список-параметров-типа*$_{opt}$
  интерфейсы-структуры$_{opt}$ *ограничения-на-параметры-типа*$_{opt}$ *тело-структуры* **;**$_{opt}$

Объявление-структуры состоит из необязательного набора *атрибутов* (раздел 17), за которыми следуют необязательный набор *модификаторов* (раздел 11.1.1), необязательный модификатор `partial`, ключевое слово `struct` и *идентификатор*, определяющий имя структуры, за которым идут необязательная спецификация *списка-параметров-типа* (раздел 10.1.3), необязательные спецификации *интерфейсов-структуры* (раздел 11.1.2) и *ограничений-на-параметры-типа* (раздел 10.1.5), за которыми следуют *тело-структуры* (раздел 11.1.4) и необязательная точка с запятой.

11.1.1. Модификаторы структур

Объявление-структуры может включать необязательную последовательность модификаторов:

модификаторы-структуры:
  модификатор-структуры
  модификаторы-структуры модификатор-структуры

модификатор-структуры:
  new
  public
  protected
  internal
  private

Если один и тот же модификатор встречается в объявлении структуры несколько раз, возникает ошибка компиляции.

Модификаторы в объявлении структуры имеют то же значение, что и в объявлении класса (раздел 10.1).

11.1.2. Модификатор partial

Модификатор `partial` указывает на то, что *объявление-структуры* является частичным объявлением. Несколько частичных объявлений, находящихся в одном пространстве имен или объявлении типа, объединяются в одно объявление в соответствии с правилами, указанными в разделе 10.2.

11.1.3. Интерфейсы структуры

Объявление структуры может включать в себя спецификацию *интерфейсов-структуры*, в этом случае структура непосредственно реализует указанные интерфейсы.

```
интерфейсы-структуры:
   :   список-интерфейсов
```

Реализация интерфейсов рассматривается в разделе 13.4.

11.1.4. Тело структуры

Тело-структуры определяет список элементов структуры.

```
тело-структуры:
   {   объявления-элементов-структуры_opt   }
```

11.2. Элементы структуры

Элементы структуры состоят из элементов, перечисленных в *объявлениях-элементов-структуры*, и элементов, унаследованных от `System.ValueType`.

```
объявления-элементов-структуры:
   объявление-элемента-структуры
   объявления-элементов-структуры   объявление-элемента-структуры

объявление-элемента-структуры:
   объявление-константы
   объявление-поля
   объявление-метода
   объявление-свойства
   объявление-события
   объявление-индексатора
   объявление-операции
   объявление-конструктора
   объявление-статического-конструктора
   объявление-типа
```

Не считая различий, указанных в разделе 11.3, описание элементов класса, приведенное в разделах 10.3–10.14, подходит и для структур.

11.3. Различия между классами и структурами

Существуют несколько важных отличий структур от классов:

- Структуры являются типами-значениями (раздел 11.3.1).
- Все структурные типы неявно наследуются от класса `System.ValueType` (раздел 11.3.2).
- Присваивание переменной структурного типа создает *копию* присваиваемого значения (раздел 11.3.3).
- Значение по умолчанию для структуры — это значение, получившееся в результате присваивания всем ее полям, имеющим типы-значения, их значений по умолчанию, а имеющим ссылочные типы — `null` (раздел 11.3.4).
- Для преобразования `struct` в `object` и обратно используются операции упаковки и распаковки (раздел 11.3.5).
- В структурах ключевое слово `this` имеет другое значение (раздел 7.6.7).
- Объявления полей структуры не могут содержать инициализаторы переменных (раздел 11.3.7).
- В структурах нельзя объявлять конструкторы без параметров (раздел 11.3.8).
- В структурах нельзя объявлять деструкторы (раздел 11.3.9).

ДЖЕСС ЛИБЕРТИ

Обратите внимание, что приведенные различия семантически значимы, в отличие от незначительных различий между структурами и классами в C++.

11.3.1. Значимая семантика

Структуры являются типами-значениями (раздел 4.1), и принято говорить, что они имеют значимую семантику. Классы, напротив, являются ссылочными типами (раздел 4.2) и имеют ссылочную семантику.

БИЛЛ ВАГНЕР

Приведенное ниже утверждение о том, что переменная структурного типа непосредственно содержит данные, входит в число тех утверждений из спецификации, которые, хоть и являются корректными, могут ввести в заблуждение. Структура может содержать элементы ссылочных типов. Данные, содержащиеся в структуре, являются ссылкой на тип класса. Например:

```
struct Message
{
   int code;
   string message;
```

```
   // переменная message содержит ссылку на объект
   // типа string, а не сами символы
}
```

Операция присваивания копирует ссылку на `message`, а не сами символы.

Переменная типа `struct` непосредственно содержит данные структуры, в то время как переменная типа класса содержит ссылку на данные, известную как объект. Если структура `B` содержит поле экземпляра типа `A`, и `A` имеет структурный тип, возникает ошибка компиляции, если `A` зависит от `B`. Структура `X` *непосредственно зависит* от структуры `Y`, если `X` содержит поле типа `Y`. С учетом этого определения, полный набор структур, от которых зависит данная структура — это транзитивное замыкание отношения *непосредственно зависит от*. Например, объявление

```
struct Node
{
   int data;
   Node next; // Ошибка: Node непосредственно зависит от себя
}
```

является ошибочным, поскольку `Node` содержит поле экземпляра типа `Node`. Другой пример

```
struct A { B b; }
struct B { C c; }
struct C { A a; }
```

тоже содержит ошибку, так как все типы `A`, `B` и `C` зависят друг от друга.

При использовании классов допустимо, что две переменные ссылаются на один и тот же объект, поэтому операции над одной переменной могут изменять объект, на который ссылается другая переменная. В случае со структурами каждая переменная содержит собственную копию данных (за исключением переменных, передаваемых в качестве параметров с ключевыми словами `ref` и `out`), и операции над одной переменной не могут влиять на другие. Более того, поскольку структуры не являются ссылочными типами, переменная структурного типа не может принимать значение `null`.

Рассмотрим объявление

```
struct Point
{
   public int x, y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Результатом выполнения кода

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

будет `10`. Присваивание переменной `b` значения переменной `a` создает копию значения, поэтому `b` не изменяется при изменении `a.x`. Если бы `Point` являлась классом, этот код вывел бы `100`, поскольку `a` и `b` ссылались бы на один и тот же объект.

11.3.2. Наследование

Все структурные типы неявно унаследованы от класса `System.ValueType`, который, в свою очередь, унаследован от класса `object`. Объявление структуры может включать в себя список реализуемых ей интерфейсов, но указать базовый класс в объявлении структуры невозможно.

Структурные типы не могут быть абстрактными, и от них нельзя наследовать. Следовательно, в объявлении структуры недопустимо использование модификаторов `abstract` и `sealed` (последний подразумевается неявно).

В связи с невозможностью наследовать от структуры к ее элементам не применимы модификаторы доступа `protected` и `protected internal`.

Функциональные элементы структуры нельзя объявлять с модификаторами `abstract` или `virtual`, а модификатор `override` можно использовать только для переопределения методов, унаследованных от `System.ValueType`.

11.3.3. Присваивание

Присваивание значения переменной типа `struct` создает *копию* присваиваемого значения. В отличие от структур выполнение операции присваивания над переменной типа класса приводит к копированию ссылки, а не объекта, на который она ссылается.

Аналогично присваиванию, при передаче структуры в метод в качестве параметра или возврате ее из метода в качестве результата создается копия структуры. Чтобы передать структуру по ссылке, можно использовать модификаторы параметров `ref` или `out`.

> **ЭРИК ЛИППЕРТ**
>
> Использование модификаторов `ref` или `out` можно рассматривать как создание псевдонима для переменных. Вместо того чтобы думать: «Я собираюсь использовать модификатор `ref` для передачи этой структуры по ссылке», — я предпочитаю думать: «Я собираюсь использовать модификатор `ref`, чтобы сделать этот параметр псевдонимом для переменной, содержащей данную структуру».

Когда выполняется присваивание индексатору или свойству структуры, выражение экземпляра, которое соответствует этому свойству или индексатору, должно классифицироваться как переменная. Если оно классифицируется как значение, возникает ошибка компиляции. Более подробно это описано в разделе 7.17.1.

11.3.4. Значения по умолчанию

Как было сказано в разделе 5.2, определенные типы переменных при создании автоматически инициализируются значениями по умолчанию. Для переменных типа класса и других ссылочных типов этим значением является `null`. Однако так как структуры являются типами-значениями и не могут быть `null`, значение по умолчанию для структуры — это значение, получившееся в результате присваивания всем ее полям, имеющим типы-значения, их значений по умолчанию, а имеющим ссылочные типы — `null`.

Для структуры `Point`, объявление которой приведено выше, код

```
Point[] a = new Point[100];
```

инициализирует каждый объект `Point` в массиве значением, получившимся в результате инициализации нулями полей `x` и `y`.

Значение по умолчанию для структуры соответствует значению, возвращаемому ее конструктором по умолчанию (раздел 4.1.2). В отличие от класс в структуре нельзя объявлять конструктор экземпляра без параметров. Вместо этого в каждой структуре присутствует неявный конструктор экземпляра без параметров, который всегда возвращает значение, получаемое путем присваивания всем полям, имеющим типы-значения, их значений по умолчанию, а имеющим ссылочные типы — `null`.

Структуры необходимо проектировать таким образом, чтобы их состояние после инициализации по умолчанию можно было считать корректным. В примере

```
using System;

struct KeyValuePair
{
   string key;
   string value;

   public KeyValuePair(string key, string value) {
      if (key == null || value == null)
         throw new ArgumentException();
      this.key = key;
      this.value = value;
   }
}
```

определенный пользователем конструктор экземпляра не допускает присваивания полям `key` и `value` значений `null` только в том случае, когда он вызывается явно. Если переменная типа `KeyValuePair` инициализируется по умолчанию, значениями этих полей будет `null`, и в структуре должно быть учтено такое состояние.

11.3.5. Упаковка и распаковка

> **БИЛЛ ВАГНЕР**
>
> Начиная с C# 2.0, у вас зачастую есть возможность избежать использования упаковки и распаковки, используя обобщения.

Значение типа класса можно преобразовать в тип `object` или в тип интерфейса, реализованного классом, просто обрабатывая при компиляции ссылку как другой тип.

Точно так же можно осуществить обратное преобразование значения типа `object` или типа интерфейса в тип класса без изменения ссылки (разумеется, в этом случае требуется проверка на этапе выполнения программы).

Так как структуры не являются ссылочными типами, вышеописанные операции для них выполняются по-другому. Когда значение структурного типа преобразуется в тип `object` или тип интерфейса, реализованного структурой, выполняется операция упаковки. Соответственно, при преобразовании значения типа `object` или типа интерфейса обратно в структурный тип выполняется операция распаковки. Ключевое отличие этих операций от аналогичных операций над классами заключается в том, что упаковка и распаковка осуществляют *копирование* значения структуры в упакованный экземпляр или из него. Таким образом, после этих операций никакие изменения в распакованной структуре не отразятся на упакованной.

ЭРИК ЛИППЕРТ

Вот еще одна причина, по которой типы-значения должны быть неизменяемы: если изменения невозможны, то факт, что изменения в распакованной структуре не влияют на упакованную, не важен. Вместо того чтобы разбираться в неожиданной и запутанной семантике, лучше ее избегать.

Когда в структурном типе переопределен виртуальный метод, унаследованный от `System.Object` (например, `Equals`, `GetHashCode` или `ToString`), вызов этого метода через экземпляр структуры не приводит к операции упаковки. Это верно, даже если структура используется как параметр типа и вызов осуществляется через экземпляр этого типа.

Например:

```
using System;

struct Counter
{
   int value;

   public override string ToString() {
      value++;
      return value.ToString();
   }
}

class Program
{
   static void Test<T>() where T: new() {
      T x = new T();
      Console.WriteLine(x.ToString());
      Console.WriteLine(x.ToString());
```

```
      Console.WriteLine(x.ToString());
   }
   static void Main() {
      Test<Counter>();
   }
}
```

Эта программа выводит

```
1
2
3
```

Хотя наличие побочных эффектов у метода `ToString` — признак плохого стиля, данный пример демонстрирует, что ни при одном из трех вызовов `x.ToString()` не выполнялась операция упаковки.

Неявная упаковка также никогда не происходит при доступе к элементу ограниченного параметра-типа . Для примера рассмотрим интерфейс `ICounter`, содержащий метод `Increment`, который может быть использован для изменения значения. Если `ICounter` используется в качестве ограничения параметра-типа, при вызове метода `Increment` ему передается ссылка на переменную, через которую он был вызван, а не упакованная копия.

```
using System;

interface ICounter
{
   void Increment();
}

struct Counter: ICounter
{
   int value;

   public override string ToString() {
      return value.ToString();
   }

   void ICounter.Increment() {
      value++;
   }
}

class Program
{
   static void Test<T>() where T: ICounter, new() {
      T x = new T();
      Console.WriteLine(x);
      x.Increment();
   // Изменяем x
      Console.WriteLine(x);
      ((ICounter)x).Increment();
   // Изменяем упакованную копию x
      Console.WriteLine(x);
   }
```

продолжение ⇗

```
   static void Main() {
      Test<Counter>();
   }
}
```

Первый вызов `Increment` изменяет значение переменной `x`. Он отличается от второго вызова `Increment`, который изменяет значение упакованной копии `x`. Таким образом, программа выведет

```
0
1
1
```

Более подробно об упаковке и распаковке написано в разделе 4.3.

11.3.6. Значение this

Внутри конструктора или функции экземпляра класса `this` классифицируется как значение. Таким образом, хотя `this` и может использоваться, чтобы ссылаться на экземпляр, через который был вызван функциональный элемент, внутри этого элемента `this` нельзя ничего присвоить.

Внутри конструктора экземпляра структуры `this` соответствует параметру `out` структурного типа; внутри функции экземпляра структуры `this` соответствует параметру `ref` структурного типа. В обоих случаях `this` классифицируется как переменная, что делает возможным изменение всей структуры, для которой был вызван функциональный элемент, путем присваивания `this` какого-либо значения или передачи `this` в качестве параметра `ref` или `out`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
> Анонимные функции и выражения-запросы внутри структур не имеют доступа к `this` или элементам экземпляра `this`.

11.3.7. Инициализаторы полей

Как говорилось в разделе 11.3.4, значение по умолчанию для структуры представляет собой значение, полученное в результате присваивания всем полям, имеющим типы-значения, их значений по умолчанию, а имеющим ссылочные типы — `null`. По этой причине в структуре не допускается инициализация полей экземпляра при объявлении. Это ограничение применяется только к полям экземпляра. Объявления статических полей структуры могут включать инициализаторы. Пример

```
struct Point
{
   public int x = 1; // Ошибка: использование инициализатора запрещено
   public int y = 1; // Ошибка: использование инициализатора запрещено
}
```

является ошибочным, поскольку объявления полей экземпляра содержат инициализаторы.

ДЖОЗЕФ АЛЬБАХАРИ

Следующий пример показывает, как можно обойти это ограничение и установить 1 в качестве значения по умолчанию для `x`:

```
struct Point
{
   bool initialized; // Значение по умолчанию — false
   int x;
   public int X {
      get {
         if (!initialized) { x = 1; initialized = true; }
         return x;
      }
   }
}
```

ЭРИК ЛИППЕРТ

Для более компактной записи предложенного Джозефом варианта можно спрятать флаг в обнуляемом типе и использовать операцию нулевого объединения:

```
struct Point
{
   private int? x; // Значение по умолчанию — null
   public int X { get { return x ?? 1; }}
}
```

Так как обнуляемый `int` на самом деле реализован в виде структуры, содержащей `int` и `bool`, это, в сущности, та же самая техника, только в более сжатом виде.

11.3.8. Конструкторы

В отличие от класса в структуре нельзя объявлять конструктор экземпляра без параметров. Вместо этого в каждой структуре присутствует неявный конструктор экземпляра без параметров, который всегда возвращает значение, получаемое путем присваивания всем полям, имеющим типы-значения, их значений по умолчанию, а имеющим ссылочные типы — `null` (раздел 4.1.2). В структуре могут быть объявлены конструкторы экземпляра с параметрами. Например:

```
struct Point
{
   int x, y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Для приведенного объявления операторы

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

создают экземпляр `Point`, в котором `x` и `y` инициализированы нулями.

Конструктор экземпляра структуры не может содержать инициализатор конструктора базового класса в форме `base(…)`.

Если объявление конструктора экземпляра структуры не содержит инициализатор конструктора, переменная `this` соответствует параметру `out` структурного типа; как и для любого другого параметра `out`, `this` должен быть явно присвоен (раздел 5.3) в каждой точке возврата из конструктора. Если объявление конструктора экземпляра структуры содержит инициализатор конструктора, переменная `this` соответствует параметру `ref` структурного типа; как и для любого другого параметра `ref`, предполагается, что при входе в конструктор `this` является явно присвоенным. Рассмотрим реализацию конструктора экземпляра:

```
struct Point
{
   int x, y;
   public int X {
      set { x = value; }
   }
   public int Y {
      set { y = value; }
   }
   public Point(int x, int y) {
      X = x; // Ошибка: this еще не является явно присвоенным
      Y = y; // Ошибка: this еще не является явно присвоенным
   }
}
```

Ни одна из функций экземпляра (включая части `set` для свойств `X` и `Y`) не может быть вызвана до тех пор, пока все создаваемые поля структуры не будут явно присвоенными. Стоит, однако, заметить, что если бы `Point` был классом, а не структурой, такая реализация конструктора экземпляра была бы допустима.

ЭРИК ЛИППЕРТ

В спецификации не указано, что заставить приведенный код работать достаточно просто — нужно вызвать конструктор по умолчанию, чтобы гарантировать инициализацию всех полей:

```
public Point(int x, int y) : this() { // В этом случае структура будет
                                      // проинициализирована
```

11.3.9. Деструкторы

Объявление деструктора в структуре недопустимо.

11.3.10. Статические конструкторы

К статическим конструкторам структур применимо большинство аналогичных правил для классов. Причиной вызова статического конструктора для структур-

ного типа является то из следующих событий в приложении, которое произойдет первым:

- Обращение к статическому элементу структуры.
- Вызов явно объявленного конструктора структурного типа.

Создание значений по умолчанию (раздел 11.3.4) в структурных типах не приводит к вызову статического конструктора (пример — начальные значения элементов массива).

11.4. Примеры структур

Ниже приведены два важных примера использования типов `struct` для создания типов, которые могут быть использованы почти так же, как встроенные типы, но с измененной семантикой.

БИЛЛ ВАГНЕР

В связи с добавлением обнуляемых типов в язык и платформу эти примеры уже не выглядят столь убедительно. Тем не менее я обнаружил, что все еще использую типы `struct` для хранения экземпляров типов при создании очень больших коллекций данных.

11.4.1. Целочисленный тип для базы данных

Приведенная ниже структура `DBInt` реализует целочисленный тип, способный представлять все множество значений типа `int`, а также дополнительное состояние, обозначающее неизвестное значение. Тип с такими характеристиками широко используется в базах данных.

```
using System;

public struct DBInt
{
   // Элемент Null представляет неизвестное значение DBInt.

   public static readonly DBInt Null = new DBInt();

   // Когда поле defined имеет значение true, DBInt представляет
   // известное значение, хранящееся в поле field.
   // В противном случае DBInt представляет
   // неизвестное значение, а значение поля field равно 0.

   int value;
   bool defined;

   // Закрытый конструктор экземпляра.
   // Создает DBInt с известным значением.
```

продолжение ➪

```
DBInt(int value) {
   this.value = value;
   this.defined = true;
}

// Свойство IsNull имеет значение true, если
// DBInt представляет неизвестное значение.

public bool IsNull { get { return !defined; } }

// Свойство Value содержит известное значение DBInt,
// или 0, если DBInt представляет неизвестное значение.

public int Value { get { return value; } }

// Неявное преобразование int в DBInt.

public static implicit operator DBInt(int x) {
   return new DBInt(x);
}

// Явное преобразование DBInt в int. Выбрасывает исключение,
// если переданное DBInt содержит неизвестное значение.

public static explicit operator int(DBInt x) {
   if (!x.defined) throw new InvalidOperationException();
   return x.value;
}

public static DBInt operator +(DBInt x) {
   return x;
}

public static DBInt operator -( DBInt x) {
   return x.defined ? -x.value : Null;
}

public static DBInt operator +( DBInt x, DBInt y) {
   return x.defined && y.defined? x.value + y.value: Null;
}

public static DBInt operator -( DBInt x, DBInt y) {
   return x.defined && y.defined? x.value - y.value: Null;
}

public static DBInt operator *( DBInt x, DBInt y) {
   return x.defined && y.defined? x.value * y.value: Null;
}

public static DBInt operator /(DBInt x, DBInt y) {
   return x.defined && y.defined? x.value / y.value: Null;
}

public static DBInt operator %(DBInt x, DBInt y) {
   return x.defined && y.defined? x.value % y.value: Null;
}
```

```
   public static DBBool operator ==(DBInt x, DBInt y) {
      return x.defined &&
               y.defined? x.value == y.value: DBBool.Null;
   }

   public static DBBool operator !=(DBInt x, DBInt y) {
      return x.defined &&
               y.defined? x.value != y.value: DBBool.Null;
   }

   public static DBBool operator >(DBInt x, DBInt y) {
      return x.defined &&
               y.defined? x.value > y.value: DBBool.Null;
   }

   public static DBBool operator <(DBInt x, DBInt y) {
      return x.defined &&
               y.defined? x.value < y.value: DBBool.Null;
   }

   public static DBBool operator >=(DBInt x, DBInt y) {
      return x.defined &&
               y.defined? x.value >= y.value: DBBool.Null;
   }

   public static DBBool operator <=(DBInt x, DBInt y) {
      return x.defined &&
               y.defined? x.value <= y.value: DBBool.Null;
   }

   public override bool Equals(object obj) {
      if (!(obj is DBInt)) return false;
      DBInt x = (DBInt)obj;
      return value == x.value && defined == x.defined;
   }

   public override int GetHashCode() {
      return value;
   }

   public override string ToString() {
      return defined? value.ToString():"DBInt.Null";
   }
}
```

КРИСТИАН НЕЙГЕЛ

В пространстве имен `System.Data.SqlTypes` платформы .NET содержатся отображающие структуры, такие как `SqlBoolean` и `SqlInt32`, которые в чем-то схожи с представленными здесь `DBInt` и `DBBool`. Эти типы данных доступны, начиная с .NET 1.0, и были созданы, чтобы обеспечить хранение значения `null`, поскольку оно допустимо для типов, используемых в базах данных. В .NET 2.0 появились обнуляемые типы, поэтому создания собственных типов можно избежать и использовать вместо них `int?` и `bool?`.

11.4.2. Тип boolean для базы данных

Приведенная ниже структура `DBBool` реализует логический тип с тремя допустимыми значениями: `DBBool.True`, `DBBool.False` и `DBBool.Null`, где элемент `Null` представляет неизвестное значение. Такие типы с тремя значениями повсеместно используются в базах данных.

```
using System;
public struct DBBool
{
   // Три возможных значения DBBool.

   public static readonly DBBool Null = new DBBool(0);
   public static readonly DBBool False = new DBBool(-1);
   public static readonly DBBool True = new DBBool(1);

   // Закрытое поле, которое хранит –1, 0, 1
   // для False, Null и True соответственно.

   sbyte value;

   // Закрытый конструктор экземпляра.
   // Параметр value должен иметь значение –1, 0 или 1.

   DBBool(int value) {
      this.value = (sbyte)value;
   }

   // Свойства, используемые для проверки значения DBBool.
   // Возвращают true, если DBBool имеет указанное значение,
   // иначе — false.

   public bool IsNull { get { return value == 0; } }
   public bool IsFalse { get { return value < 0; } }
   public bool IsTrue { get { return value > 0; } }

   // Неявное преобразование bool в DBBool. Преобразует true
   // в DBBool.True, а false — в DBBool.False.

   public static implicit operator DBBool(bool x) {
      return x? True: False;
   }

   // Явное преобразование bool в DBBool.
   // Выбрасывает исключение, если переданное DBBool
   // имеет значение Null; иначе возвращает true или false.

   public static explicit operator bool(DBBool x) {
      if (x.value == 0) throw new InvalidOperationException();
      return x.value > 0;
   }

   // Операция проверки на равенство. Возвращает Null,
   // если хотя бы один операнд имеет значение Null;
   // иначе возвращает True или False.
```

```
public static DBBool operator ==(DBBool x, DBBool y) {
   if (x.value == 0 || y.value == 0) return Null;
   return x.value == y.value? True: False;
}

// Операция проверки на неравенство. Возвращает Null,
// если хотя бы один операнд имеет значение Null;
// иначе возвращает True или False.

public static DBBool operator !=(DBBool x, DBBool y) {
   if (x.value == 0 || y.value == 0) return Null;
   return x.value != y.value? True: False;
}

// Операция логического отрицания. Возвращает True,
// если операнд имеет значение False,
// Null, если операнд имеет значение Null,
// и False, если значение операнда равно True.

public static DBBool operator !(DBBool x) {
   return new DBBool(-x.value);
}

// Операция логического AND. Возвращает False, если
// хотя бы один из операндов имеет значение False;
// иначе возвращает Null,
// если хотя бы один из операндов
// имеет значение Null;
// в остальных случаях возвращает True.

public static DBBool operator &(DBBool x, DBBool y) {
   return new DBBool(x.value < y.value? x.value: y.value);
}

// Операция логического OR. Возвращает True, если
// хотя бы один из операндов имеет значение True;
// иначе возвращает Null,
// если хотя бы один из операндов
// имеет значение Null;
// в остальных случаях возвращает False.

public static DBBool operator |(DBBool x, DBBool y) {
   return new DBBool(x.value > y.value? x.value: y.value);
}

// Операция проверки на истинность. Возвращает true,
// если операнд имеет значение True,
// в противном случае возвращает false.

public static bool operator true(DBBool x) {
   return x.value > 0;
}

// Операция проверки на ложность. Возвращает true,
// если операнд имеет значение False,
// в противном случае возвращает false.
```

продолжение ⇗

```
   public static bool operator false(DBBool x) {
      return x.value < 0;
   }

   public override bool Equals(object obj) {
      if (!(obj is DBBool)) return false;
      return value == ((DBBool)obj).value;
   }

   public override int GetHashCode() {
      return value;
   }

   public override string ToString() {
      if (value > 0) return "DBBool.True";
      if (value < 0) return "DBBool.False";
      return "DBBool.Null";
   }
}
```

КРИС СЕЛЛЗ

Когда платформа .NET только появилась, определяемые пользователем типы-значения были ее важным отличием от Java. На практике эти типы в основном используются для оптимизации, если в процессе профилирования обнаруживается, что сборщик мусора перегружен лишней работой.

ЭРИК ЛИППЕРТ

Тот факт, что некоторые локальные переменные типа-значения могут быть без особых затрат расположены в стеке, не означает, что они автоматически становятся «более производительными», чем ссылочные типы. Размещение в куче в некотором смысле более затратно, чем размещение в стеке, но разница не так уж и велика. Существенные различия становятся заметны при освобождении памяти: чем меньше объектов было размещено в куче, тем меньше мусора понадобится идентифицировать и собрать сборщику мусора. Однако даже в этом случае любая экономия при выделении и освобождении памяти часто сводится на нет необходимостью копирования типов-значений по значению. Процессоры оптимизированы для копирования объектов, размер которых приблизительно равен размеру ссылки. Типы-значения могут иметь любой размер и часто копироваться, что в некоторых случаях увеличивает суммарные затраты вычислительных ресурсов.

Я рекомендую делать выбор между типами-значениями и ссылочными типами, основываясь на том, имеет ли больший смысл копировать по значению или по ссылке. Затем необходимо протестировать производительность средствами профилирования. Заменяйте ссылочный тип на тип-значение (или наоборот) только в том случае, если у вас есть эмпирические воспроизводимые данные, указывающие на то, что подобные изменения приведут к измеримым и существенным различиям.

Глава 12
Массивы

Массив представляет собой структуру данных, содержащую некоторое количество переменных, доступ к которым осуществляется по вычисляемым индексам. Переменные, содержащиеся в массиве, называются его элементами и имеют один и тот же тип, который называется типом элементов массива.

Массив имеет размерность, которая определяет количество индексов, связанных с элементом массива. Размерность массива часто называют его измерениями. Массив размерности 1 называют **одномерным**. Если размерность массива больше 1, массив называют **многомерным**. В зависимости от размерности многомерные массивы называют двумерными, трехмерными и т. д.

Каждое измерение массива имеет определенную длину, которая является целым числом, большим либо равным нулю. Длины измерений массива не являются частью типа этого массива, а устанавливаются во время выполнения при создании экземпляра массива. Длина измерения определяет допустимый диапазон индексов для данного измерения: для размерности длиной `N` диапазон индексов будет от `0` до `N-1` включительно. Общее количество элементов в массиве равно произведению длин всех измерений массива. Если одно или несколько измерений массива имеют нулевую длину, массив является пустым.

Тип элемента массива может быть любым, в том числе и массивом.

12.1. Типы массивов

Тип массива описывается как тип, не являющийся массивом, за которым следует один или более *спецификаторов-размерности*:

```
массив:
   не-массив    спецификаторы-размерности

не-массив:
   тип

спецификаторы-размерности:
   спецификатор-размерности
   спецификаторы-размерности    спецификатор-размерности

спецификатор-размерности:
   [    разделители-размерности_opt   ]

разделители-размерности:
   ,
   разделители-размерности    ,
```

Не-массив — это любой тип, который не является *массивом*.

Размерность массива задается крайним левым *спецификатором-размерности* в *типе-массива*: в *спецификаторе-размерности* задается, что размерность массива равна 1 плюс количество «,» обозначенных в *спецификаторе-размерности*.

Тип элемента массива можно узнать в результате удаления крайнего левого *спецификатора-размерности*:

- Тип массива в форме `T[R]` является массивом размерности `R` и элементами типа `T`, не являющегося массивом.
- Тип массива в форме `T[R][R1]...[Rn]` является массивом размерности `R` и элементами типа `T[R1]...[Rn]`.

По сути, *спецификаторы-размерности* читаются слева *до* последнего элемента типа, не являющегося массивом. Тип `int[][,,][,]` — это одномерный массив, состоящий из трехмерных массивов, которые, в свою очередь, состоят из двумерных массивов типа `int`.

> **ЭРИК ЛИППЕРТ**
>
> То, что спецификаторы размерности массивов читаются «назад», часто путает людей. Но плюсом такой схемы является то, что индексация операций идет в том же порядке, что и объявление; вы индексируете этот сложный массив как `arr[a][b,c,d][e,f]`.

Во время выполнения программы значение типа-массива может быть `null` или ссылкой на экземпляр типа этого массива.

12.1.1. Тип System.Array

Тип `System.Array` — абстрактный базовый тип всех массивов. Существуют неявное ссылочное преобразование (раздел 6.1.6) из любого массива к типу `System.Array` и явное ссылочное преобразование (раздел 6.2.4) из `System.Array` в любой тип массива. Однако `System.Array` сам по себе не является *массивом*. Напротив, это *класс*, производными от которого являются все *массивы*.

Во время выполнения программы значение типа `System.Array` может быть `null` или ссылкой на экземпляр массива любого типа.

12.1.2. Массивы и обобщенный интерфейс IList

Одномерный массив `T[]` реализует интерфейс `System.Collections.Generic.IList<T>` (для краткости `IList<T>`) и его базовые интерфейсы. Соответственно существует неявное преобразование из `T[]` к `IList<T>` и его базовые интерфейсы. Вдобавок к этому, если существует неявное ссылочное преобразование из `S` к `T`, то `S[]` реализует `IList<T>` и существует неявное ссылочное преобразование из `S[]` к `IList<T>` и его базовым интерфейсам (раздел 6.1.6). Если существует

явное ссылочное преобразование из `S` к `T`, то существует и явное ссылочное преобразование из `S[]` к `IList<T>` и его базовым интерфейсам (раздел 6.2.4). Например:

```
using System.Collections.Generic;
class Test
{
    static void Main() {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;
        IList<string> lst1 = sa;                  // Правильно
        IList<string> lst2 = oa1;      // Ошибка: необходимо приведение типа
        IList<object> lst3 = sa;                  // Правильно
        IList<object> lst4 = oa1;                 // Правильно
        IList<string> lst5 = (IList<string>)oa1;  // Исключение
        IList<string> lst6 = (IList<string>)oa2;  // Правильно
    }
}
```

Присваивание `lst2 = oa1` вызывает ошибку компиляции, потому что преобразование `object[]` в `IList<string>` является явным, а не неявным. Приведение `(IList<string>)oa1` является причиной исключения во время выполнения, потому что `oa1` ссылается на `object[]`, а не на `string[]`. Однако приведение `(IList<string>)oa2` не будет являться причиной возникновения исключения, поскольку `oa2` ссылается на `string[]`.

Если существует неявное или явное ссылочное преобразование из `S[]` к `IList<T>`, существует также явное ссылочное преобразование из `IList<T>` и его базовых интерфейсов к `S[]` (раздел 6.2.4).

Когда массив типа `S[]` реализует `IList<T>`, некоторые элементы реализованных интерфейсов могут вызывать исключения. Точное поведение реализаций интерфейса лежит вне сферы данной спецификации.

ЭРИК ЛИППЕРТ

Занятно рассмотреть нереальный мир, в котором CLR версии 1 уже имела бы обобщенные типы. В этом мире, возможно, существовали бы обобщенные типы, скажем, `Array<T>`, `Array2<T>` и т. д. И объявления более сложных типов массивов выглядели бы более понятно: `Array<Array2<int>>` — одномерный массив, где каждый элемент есть двумерный массив из целых чисел.

БИЛЛ ВАГНЕР

Тот факт, что массив `T[]` реализует `IList<T>`, выбрасывая исключения для некоторых методов, является одной из тех проблем реального мира, для которых не существует хорошего решения. `IList<T>` имеет методы добавления или удаления элементов. Массивы, однако, имеют фиксированный размер и не могут поддерживать некоторые методы. Но при этом вы хотите получить произвольный доступ к элементам, который существует в интерфейсе `IList<T>`.

12.2. Создание массива

Экземпляры массива создаются с помощью *выражений-создания-массива* (раздел 7.6.10.4) или через объявления полей и локальных переменных, содержащих *инициализатор-массива* (раздел 12.6).

При создании экземпляра массива задаются его размерность и длина каждого измерения, которые остаются неизменными на протяжении всего времени жизни экземпляра. Иными словами, впоследствии невозможно изменить ни размерность существующего экземпляра массива, ни его длину.

Экземпляр массива всегда имеет тот же тип, что и тип массива. `System.Array` — абстрактный тип, который не может быть инстанцирован.

Элементы массивов, созданных с помощью *выражения-создания-массива,* всегда инициализируются их значениями по умолчанию (раздел 5.2).

12.3. Доступ к элементам массива

Доступ к элементам массива осуществляется путем использования выражения *доступ-к-элементу-массива* (раздел 7.6.6.1) в форме `A[I1, I2, ..., In]`, где `A` — выражение типа массива и каждый I_x — выражение типа `int, uint, long` или `ulong` или может быть неявно преобразовано к любому из этих типов. Результатом доступа к элементам будет переменная, а именно элемент массива, выбранный с помощью индексов.

Элементы массива можно перебирать с помощью оператора `foreach` (раздел 8.8.4).

12.4. Элементы типа массива

Каждый тип массива наследует элементы, объявленные в классе `System.Array`.

12.5. Ковариантность массивов

Если для двух любых *ссылочных-типов* `A` и `B` существует неявное (раздел 6.1.6) или явное (раздел 6.2.4) ссылочное преобразование из `A` к `B`, то такое же ссылочное преобразование существует из массива типа `A[R]` к массиву типа `B[R]`, где `R` — любой *спецификатор-размерности* (одинаковый для обоих массивов). Это отношение известно как **ковариантность массива**. Ковариантность массива, в частности, означает что значение массива типа `A[R]` может быть ссылкой на экземпляр массива типа `B[R]`, при условии что существует неявное ссылочное преобразование из `B` к `A`.

Из-за ковариантности массивов присваивание элементам массивов ссылочных типов включает в себя проверку во время выполнения программы, что значение, присвоенное элементу массива, имеет допустимый тип (раздел 7.17.1). Пример:

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++)
            array[i] = value;
    }
    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

Присваивание `array[i]` в методе `Fill` неявно включает проверку во время выполнения программы того, что объект, на который ссылается либо `value`, либо `null`, либо экземпляр, совместимый с фактическим типом элемента массива `array`. В `Main` первые два вызова `Fill` успешны, а третий вызов выбрасывает исключение `System.ArrayTypeMismatchException` при выполнении первого присваивания `array[i]`. Исключение возникает потому, что упакованное значение `int` не может храниться в массиве из `string`.

> **ЭРИК ЛИППЕРТ**
>
> Это мой кандидат на «худшую особенность» C#. Она допускает, что присваивание вызовет сбой во время выполнения без какого-либо намека в исходном коде на то, что подобная ошибка возможна. Это вызывает издержки производительности в широко распространенном коде лишь для того, чтобы редко используемый сценарий выполнялся быстро; доступ к массиву ссылочного типа, не являющегося бесплодным, встречается гораздо чаще, чем ковариантные преобразования массивов. Я определенно предпочитаю типобезопасную ковариантность, добавленную к `IEnumerable<T>`.

> **КРИС СЕЛЛС**
>
> Лично я использую массивы очень редко, даже одномерные, не говоря уже о многомерных или ступенчатых массивах, и эти странные особенности меня особо не волнуют. С возможностями, предоставляемыми `List<T>`, `Dictionary<K, V>` и `IEnumerable<T>`, я счастлив и вообще без массивов.

> **ПИТЕР СЕСТОФТ**
>
> Необходимость при выполнении программы проверять то, на что сетовал Эрик, происходит из ковариантности массива для типа элемента (`Student[]` будет подтипом `Person[]`, когда `Student` — подтип `Person`). Эту слабость дизайна C# делит с языком программирования Java, в котором ситуация даже хуже: из-за того, что в Java обобщенные типы реализуются с помощью очистки, не существует типа объекта, представляющего параметр-тип во время выполнения, и потому, если массив был создан как `new T[...]` для некоторого параметра-типа `T` или как `new Stack<Person>[...]`, проверку выполнить нельзя. Следовательно, в Java требуется запрещать создание массива, тип
>
> *продолжение* ☞

элементов которого — параметр-тип или тип, конструированный как экземпляр обобщенного типа. В C# параметры-типы и конструированные типы честно представлены в среде выполнения представлены, поэтому подобных ограничений на массив не существует.
В языке Scala, который содержал обобщенные типы изначально, тип массива инвариантен по отношению к типу элемента, и проверка средой выполнения не нужна (но ее все равно потребуется выполнить, если Scala компилируется в байт-код Java).

БИЛЛ ВАГНЕР

Комментарии Эрика очень хорошо обьясняют, почему так важны безопасная ковариантность и контравариантность, добавленные для обобщенных типов в C# 4.0. Улучшить одновременно и производительность, и корректность получается редко.

Ковариантность массива не относится к массивам *типов-значений*. Например, не существует преобразования, которое бы разрешало обращаться с `int[]`как с `object[]`.

12.6. Инициализаторы массива

Инициализаторы массива могут быть заданы в объявлениях поля (раздел 10.5), объявлениях локальной переменной (раздел 8.5.1) и выражениях создания массива (раздел 7.6.10.4):

```
инициализатор-массива:
   {   список-инициализаторов-переменных_opt   ,   }
   {   список-инициализаторов-переменных   ,   }

список-инициализаторов-переменных:
   инициализатор переменной
   список-инициализаторов-переменных   ,   инициализатор-переменной

инициализатор-переменной:
   выражение
   инициализатор-массива
```

Инициализатор-массива состоит из последовательности инициализаторов переменных, заключенных в символы «`{`» и «`}`» и разделенных символом «`,`». Каждый инициализатор переменной представляет собой выражение или, в случае многомерного массива, вложенный инициализатор массива.

Контекст, в котором используется инициализатор, определяет тип инициализируемого массива. В выражении создания массива тип массива непосредственно предшествует инициализатору или выводится из выражений в инициализаторе массива. При объявлении поля или переменной тип массива — это тип объявленных полей или переменных. Когда инициализатор массива используется при объявлении поля или переменной как

```
int[] a = {0, 2, 4, 6, 8};
```

это просто сокращение эквивалентного выражения создания массива:

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

Для одномерного массива инициализатор должен состоять из последовательности выражений, совместимых по присваиванию с типом элемента массива. Выражения инициализируют элементы массива по возрастанию, начиная с нулевого элемента. Количество выражений в инициализаторе массива определяет длину создаваемого экземпляра массива. Например, приведенный выше инициализатор массива создает экземпляр `int[]` длиной 5, а затем инициализирует экземпляр следующими значениями:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

Для многомерного массива инициализатор должен иметь столько уровней вложенности, сколько измерений у массива. Самый внешний уровень соответствует самому левому измерению массива, самый внутренний уровень — самому правому измерению. Длина каждого измерения массива определяется количеством элементов на соответствующем уровне вложенности инициализатора. Для каждого вложенного инициализатора массива количество элементов должно быть таким же, как у других инициализаторов массива того же уровня. Пример

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

создает двумерный массив с длиной 5 для левого измерения и длиной 2 для правого:

```
int[,] b = new int[5, 2];
```

Затем он инициализируется следующими значениями:

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

Если длина измерения, отличного от самого правого, равна 0, следующие измерения также будут иметь длину, равную 0. Пример

```
int[,] c = {};
```

создает двумерный массив с нулевой длиной как для левого, так и для правого измерений:

```
int[,] c = new int[0, 0];
```

Если выражение создания массива включает и явное задание длин измерений, и инициализатор массива, длины должны быть константными выражениями и число элементов каждого вложенного уровня должно совпадать с соответствующей длиной измерения. Вот некоторые примеры:

```
int i = 3;
int[] x = new int[3] {0, 1, 2};     // Правильно
int[] y = new int[i] {0, 1, 2};     // Ошибка: i не константа
int[] z = new int[3] {0, 1, 2, 3};  // Ошибка: длина/инициализатор не соответствуют
```

Здесь инициализатор для `y` выдает ошибку компиляции, потому что выражение длины измерения не является константой. Инициализатор для `z` также выдает ошибку компиляции из-за того, что длина и количество элементов инициализатора не совпадают.

Глава 13
Интерфейсы

Интерфейсом определяется контракт. Класс или структура, реализующие интерфейс, должны следовать условиям этого контракта. Интерфейс может наследоваться от нескольких базовых интерфейсов, а класс или структура может реализовывать несколько интерфейсов.

> **БИЛЛ ВАГНЕР**
>
> Интерфейсы лучше всего работают, когда их мало, а их области видимости имеют небольшие размеры. Чем больше интерфейс, тем больше работы потребуется проделать тем, кто будет его реализовывать. Большое количество интерфейсов увеличивает возможность возникновения двусмысленности и коллизий.

> **ЭРИК ЛИППЕРТ**
>
> Сравнение интерфейсов с контрактами, безусловно, является полезным и часто используемым. Стоит отметить, что интерфейс — довольно плохой способ представления контракта. Интерфейс лишь сообщает, какие методы доступны, какие у них имена, какие типы аргументов они принимают и какие типы значений возвращают. Никакой информации о семантике операции в контракте не представлено: не говорится о том, что объект, соответствующий этому контракту, должен уничтожаться более агрессивно, что метод `Drive()` выбрасывает исключение, если вызывается раньше метода `StartEngine()`, что первый параметр должен быть `null`, а второй — ненулевым и т. д. Вся эта информация помещается в документацию, а не куда-то, куда могут добраться инструменты для анализа. Новая система контрактов кода, появившаяся в CLR версии 4.0, позволяет вам определять условия контрактов более детально, чем это возможно посредством лишь интерфейсов, а также проводить интересный анализ этих контрактов на этапе компиляции.

Интерфейсы могут содержать методы, свойства, события и индексаторы. Сам интерфейс не предоставляет реализаций элементов, которые в нем определены. Он лишь указывает, какие элементы должны присутствовать в реализующих его классах или структурах.

13.1. Объявления интерфейсов

Объявление-интерфейса — это *объявление-типа* (раздел 9.6), которое объявляет новый тип интерфейса.

```
объявление-интерфейса:
   атрибуты_opt   модификаторы-интерфейса_opt   partial_opt   interface
        идентификатор   список-параметров-вариантного-типа_opt
        базовые-интерфейсы_opt
              ограничения-на-параметры-типа_opt   тело-интерфейса   ;_opt
```

Объявление-интерфейса состоит из необязательного набора *атрибутов* (раздел 17), за которым следуют необязательный набор *модификаторов-интерфейса* (раздел 13.1.1), необязательный модификатор `partial`, ключевое слово `interface` и *идентификатор*, определяющий имя интерфейса, за которыми идут необязательный *список-параметров-вариантного-типа* (раздел 13.1.3), необязательная спецификация *базовых-интерфейсов* (раздел 13.1.4), необязательные *ограничения-на-параметры-типа* (раздел 10.1.5), затем *тело-интерфейса* (раздел 13.1.5) и необязательная точка с запятой.

13.1.1. Модификаторы интерфейса

Объявление-интерфейса может содержать необязательную последовательность модификаторов интерфейса:

```
модификаторы-интерфейса:
   модификатор-интерфейса
   модификаторы-интерфейса   модификатор-интерфейса

модификатор-интерфейса:
   new
   public
   protected
   internal
   private
```

Если один и тот же модификатор встречается в объявлении интерфейса несколько раз, возникает ошибка компиляции.

Модификатор `new` допускается только для интерфейсов, определяемых внутри класса. Он указывает на то, что интерфейс скрывает унаследованный элемент с таким же именем, как это описано в разделе 10.3.4.

Модификаторы `public`, `protected`, `internal` и `private` управляют доступом к интерфейсу. В зависимости от контекста, в котором объявляется интерфейс, некоторые из этих модификаторов могут быть недопустимы (раздел 3.5.1).

13.1.2. Модификатор partial

Модификатор `partial` указывает на то, что *объявление-интерфейса* является частичным объявлением типа. Несколько частичных объявлений интерфейса с одинаковым именем, находящиеся в одном пространстве имен или объявлении типа, объединяются в одно объявление интерфейса в соответствии с правилами, указанными в разделе 10.2.

13.1.3. Списки параметров вариантного типа

Списки параметров вариантного типа могут использоваться только для типов интерфейсов и делегатов. От обычных *списков-параметров-типа* они отличаются наличием *аннотации-вариантности* у каждого параметра типа.

```
список-параметров-вариантного-типа:
   <   параметры-вариантного-типа   >

параметры-вариантного-типа:
   атрибуты_opt   аннотация-вариантности_opt   параметр-типа
   параметры-вариантного-типа   ,   атрибуты_opt   аннотация-вариантности_opt
   параметр-типа

аннотация-вариантности:
   in
   out
```

Если в качестве аннотации вариантности указан `out`, параметр типа называется **ковариантным**. Если в качестве аннотации вариантности указан `in`, параметр типа называется **контравариантным**. Если аннотация вариантности отсутствует, параметр типа называется **инвариантным**.

> **ЭРИК ЛИППЕРТ**
>
> Ковариантностью называется способность сохранять совместимость присваивания при сопоставлении аргумента типа обобщенному типу. Например, строка может быть присвоена переменной типа `object`. Сопоставление `T` типу `IEnumerable<T>` сохраняет совместимость присваивания; значение типа `IEnumerable<string>` может быть присвоено переменной типа `IEnumerable<object>`. Таким образом, формулировка «параметр типа ковариантен» — это сокращение; на самом деле ковариантным является отношение между аргументом типа и сконструированным типом. Корректной будет формулировка «сопоставление любого ссылочного типа `T` типу `IEnumerable<T>` ковариантно относительно `T`», но она труднопроизносима, в отличие от использования «`T` ковариантен» и понимания того, что на самом деле имелась в виду более длинная формулировка.

Пример:

```
interface C<out X, in Y, Z>
{
    X M(Y y);

    Z P { get; set; }
}
```

Здесь `X` является ковариантным, `Y` — контравариантным, а `Z` — инвариантным.

13.1.3.1. Безопасность вариантности

Наличие аннотаций вариантности в списке параметров типа ограничивает возможные позиции, в которых типы могут встречаться в объявлении типа.

Тип `T` является **небезопасным с точки зрения вывода (output-unsafe)**, если выполняется одно из следующих условий:

- `T` — контравариантный параметр типа.
- `T` — тип массива, имеющий небезопасный с точки зрения вывода тип элементов.
- `T` — тип интерфейса или делегата $S<A_1, \ldots, A_K>$, сконструированный из обобщенного типа $S<X_1, \ldots, X_K>$, где по крайней мере для одного A_i выполняется одно из следующих условий:
  - X_i ковариантен или инвариантен, и A_i является небезопасным с точки зрения вывода.
  - X_i контравариантен или инвариантен, и A_i является безопасным с точки зрения ввода (input-safe).

Тип `T` является **небезопасным с точки зрения ввода (input-unsafe)**, если выполняется одно из следующих условий:

- `T` — ковариантный параметр типа.
- `T` — тип массива, имеющий небезопасный с точки зрения ввода тип элементов.
- `T` — тип интерфейса или делегата $S<A_1, \ldots, A_K>$, сконструированный из обобщенного типа $S<X_1, \ldots, X_K>$, где по крайней мере для одного A_i выполняется одно из следующих условий:
  - X_i ковариантен или инвариантен, и A_i является небезопасным с точки зрения ввода.
  - X_i контравариантен или инвариантен, и A_i является небезопасным с точки зрения вывода.

Интуитивно понятно, что небезопасный с точки зрения вывода тип нельзя использовать в позиции вывода, а небезопасный с точки зрения ввода тип нельзя использовать в позиции ввода.

Тип является **безопасным с точки зрения вывода (output-safe)**, если он не является небезопасным с точки зрения вывода. Тип является **безопасным с точки зрения ввода (input-safe)**, если он не является небезопасным с точки зрения ввода.

13.1.3.2. Преобразование вариантности

Аннотации вариантности нужны для того, чтобы предоставить возможность осуществления менее строгих (но тем не менее типобезопасных) преобразований в типы интерфейсов и делегатов. Для достижения этой цели в определениях неявных (раздел 6.1) и явных (раздел 6.2) преобразований используется понятие преобразуемости вариантности, определяемое следующим образом:

Тип $T<A_1, \ldots, A_n>$ вариантно преобразуем в тип $T<B_1, \ldots, B_n>$, если `T` является типом интерфейса или делегата, объявленным с параметрами вариантного типа $T<X_1, \ldots, X_n>$, и для каждого параметра вариантного типа X_i выполняется одно из следующих условий:

- X_i ковариантен, и существует неявное преобразование ссылки или тождественное преобразование из A_i в B_i.

- X_i контравариантен, и существует неявное преобразование ссылки или тождественное преобразование из B_i в A_i.
- X_i инвариантен, и существует тождественное преобразование из A_i в B_i.

> **ДЖОН СКИТ**
>
> Я подозреваю, многим разработчикам никогда не потребуется объявлять свои собственные вариантные интерфейсы или делегаты. По сути, мое предположение заключается в том, что большую часть времени мы даже не будем замечать, что *используем* вариантность — просто код, работоспособность которого мы ожидаем интуитивно, теперь будет работать, даже несмотря на то, что в C# 3.0 он бы не скомпилировался.

13.1.4. Базовые интерфейсы

Интерфейс может не наследоваться ни от одного или наследоваться от одного или более интерфейсов, именуемых его **явными базовыми интерфейсами**. Когда у интерфейса существуют один или несколько явных базовых интерфейсов, в объявлении этого интерфейса за его идентификатором следуют двоеточие и список типов базовых интерфейсов, разделенных запятыми.

```
базовый-интерфейс:
    :    список-типов-интерфейсов
```

Для сконструированного типа интерфейса явные базовые интерфейсы формируются следующим образом: каждый *параметр-типа* в объявлениях базовых интерфейсов в обобщенном типе заменяется соответствующим *аргументом-типа* в сконструированном типе.

Явные базовые интерфейсы конкретного интерфейса должны иметь не более строгий вид доступа, чем у него самого (раздел 3.5.4). Например, для интерфейса с видом доступа `public` указание в числе его *базовых-интерфейсов* интерфейса с доступом `private` или `internal` приведет к ошибке компиляции.

Если интерфейс прямо или косвенно наследуется от самого себя, возникает ошибка компиляции.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Это правило игнорирует аргументы типа (если они есть). Например, несмотря на то что `I<T>` и `I<I<T>>` являются разными типами, следующее объявление все равно некорректно:
>
> ```
> interface I<T> : I<I<T>> { }
> ```
>
> И наоборот, использование интерфейса в качестве аргумента типа для его базового интерфейса является совершенно корректным:
>
> ```
> interface IA<T> { }
> interface IB : IA<IB[]> { } // Все в порядке
> ```

К **базовым интерфейсам** интерфейса относятся его явные базовые интерфейсы, а также их базовые интерфейсы. Другими словами, набор базовых интерфейсов

представляет собой полное транзитивное замыкание на множестве явных базовых интерфейсов, их базовых интерфейсов и т. д. Интерфейс наследует все элементы его базовых интерфейсов. Пример:

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Здесь базовыми интерфейсами `IComboBox` являются `IControl`, `ITextBox` и `IListBox`.

Иными словами, интерфейс `IComboBox` наследует элементы `SetText` и `SetItems`, а также `Paint`.

Каждый базовый интерфейс какого-либо интерфейса должен быть безопасен с точки зрения вывода (раздел 13.1.3.1). Класс или структура, реализующие интерфейс, также неявно реализуют все его базовые интерфейсы.

ЭРИК ЛИППЕРТ

«Наследование» — неудачно выбранное слово для интерфейсов. Обычно под наследованием от базового объекта подразумевают совместное использование реализации, что для интерфейсов, очевидно, неверно. Я предпочитаю думать об интерфейсах как о *контрактах, которые могут определять другие контракты, которые также должны быть выполнены*, а не как о контрактах, «наследующих» другие контракты.

ДЖЕСС ЛИБЕРТИ

Существует традиция добавлять к названиям интерфейсов префикс «I» (`IControl`, `IWriteable`, `IClaudius`). Хотя и не существует убедительной причины так делать, этот способ именования сохраняется уже не менее десяти лет как последнее напоминание о «венгерской» нотации. Удаление «I» в конечном счете улучшит читабельность кода.

13.1.5. Тело интерфейса

Тело-интерфейса определяет элементы интерфейса.

```
тело-интерфейса:
  {   объявления-элементов-интерфейса_opt   }
```

13.2. Элементы интерфейса

Элементы интерфейса включают в себя элементы, унаследованные от базовых интерфейсов, а также элементы, объявленные в самом интерфейсе.

объявления-элементов-интерфейса:
объявление-элементов-интерфейса
объявления-элементов-интерфейса объявление-элементов-интерфейса

объявление-элементов-интерфейса:
объявление-метода-интерфейса
объявление-свойства-интерфейса
объявление-события-интерфейса
объявление-индексатора-интерфейса

Объявление интерфейса может содержать ноль и более элементов. Элементами интерфейса должны быть методы, свойства, события или индексаторы. Интерфейс не может содержать константы, поля, операторы, конструкторы экземпляра, деструкторы или типы, а также любые статические элементы.

Все элементы интерфейса по умолчанию имеют открытый вид доступа. При указании любых модификаторов для объявлений элементов интерфейса возникает ошибка компиляции. В частности, элементы интерфейса не могут быть объявлены с модификаторами `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override` и `static`.

ДЖОН СКИТ

Иногда было бы неплохо иметь возможность создавать внутренний интерфейс с внутренними элементами, которые могли бы неявно реализовываться внутренними методами. В данный момент типы, реализующие внутренний интерфейс, должны использовать или открытые методы, или явную реализацию интерфейса; в некоторых случаях ни один из этих вариантов не является очень удобным.

Пример:

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

Здесь объявлен интерфейс, содержащий по одному из каждых возможных типов элементов: метод, свойство, событие и индексатор.

Объявление-интерфейса создает новую область объявлений (раздел 3.3), а *объявления-элементов-интерфейса*, непосредственно содержащиеся в *объявлении-интерфейса*, вводят в эту область новые элементы. К *объявлениям-элементов-интерфейса* применяются следующие правила:

- Имя метода должно отличаться от имен всех свойств и событий, объявленных в том же интерфейсе. Помимо этого, сигнатура (раздел 3.6) метода должна отличаться от сигнатур всех остальных методов, объявленных в этом интерфейсе, а два метода, объявленные в одном интерфейсе, не могут иметь сигнатуры, различающиеся только лишь модификаторами `ref` и `out`.
- Имя свойства или события должно отличаться от имен всех остальных элементов, объявленных в том же интерфейсе.
- Сигнатура индексатора должна отличаться от сигнатур всех остальных индексаторов, объявленных в том же интерфейсе.

Унаследованные элементы интерфейса не являются частью области объявлений интерфейса. Таким образом, в интерфейсе может быть объявлен элемент с таким же именем или сигнатурой, что и у унаследованного элемента. Когда это происходит, принято говорить, что унаследованный элемент интерфейса *скрывает* элемент базового интерфейса. Сокрытие унаследованного элемента не считается ошибкой, однако заставляет компилятор выдать предупреждение. Чтобы убрать это предупреждение, необходимо добавить к объявлению унаследованного элемента модификатор `new`, означающий, что сокрытие унаследованным элементом базового является намеренным. Более подробно этот вопрос рассматривается в разделе 3.7.1.2.

Если модификатор `new` используется в объявлении, которое не скрывает унаследованный элемент, компилятор выдает предупреждение. Его можно убрать, удалив модификатор `new`.

Заметьте, что элементы класса `object`, строго говоря, не являются элементами какого-либо интерфейса (раздел 13.2). Однако элементы этого класса доступны через поиск элементов в любом типе интерфейса (раздел 7.4).

13.2.1. Методы интерфейса

Методы интерфейса объявляются через *объявления-методов-интерфейса*:

объявление-метода-интерфейса:
   атрибуты$_{opt}$ **new**$_{opt}$ *тип-возвращаемого-значения идентификатор*
   список-параметров-типа **(** *список-формальных-параметров*$_{opt}$ **)**
   ограничения-на-параметры-типа$_{opt}$ **;**

Атрибуты, *тип-возвращаемого-значения*, *идентификатор* и *список-формальных-параметров* в объявлении метода интерфейса имеют то же значение, что и в объявлении метода класса (раздел 10.6). В объявлении метода интерфейса не допускается указывать тело метода, поэтому объявление всегда оканчивается точкой с запятой.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> В отличие от объявлений методов в классах или структурах, *идентификатор* в *объявлении-метода-интерфейса* может быть таким же, как имя содержащего его интерфейса. Это верно также для *объявлений-свойств-интерфейса* и *объявлений-событий-интерфейса*.

Тип каждого формального параметра метода интерфейса должен быть безопасным с точки зрения ввода (раздел 13.1.3.1), а возвращаемое значение должно быть

безопасным с точки зрения вывода или `void`. Более того, каждое ограничение на тип класса, на тип интерфейса и на параметр типа для каждого параметра типа в методе должно быть безопасным с точки зрения ввода.

Эти правила гарантируют, что любое ковариантное или контравариантное использование интерфейса остается типобезопасным. Например, код

```
interface <I<out T> { void M<U>() where U : T; }
```

некорректен, поскольку использование `T` в качестве ограничения на параметр типа для `U` не является безопасным с точки зрения ввода.

Если бы такого ограничения не существовало, было бы возможным нарушить типобезопасность следующим образом:

```
class B {}
class D : B {}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

Фактически это вызов `C.M<E>`. Однако этот вызов требует, чтобы `E` наследовался от `D`, следовательно, в данном случае типобезопасность была бы нарушена.

13.2.2. Свойства интерфейса

Свойства интерфейса объявляются через *объявления-свойств-интерфейса*:

объявление-свойства-интерфейса:
  атрибуты$_{opt}$ **new**$_{opt}$ *тип* *идентификатор* **{** *коды-доступа-интерфейса* **}**

коды-доступа-интерфейса:
  атрибуты$_{opt}$ **get** **;**
  атрибуты$_{opt}$ **set** **;**
  атрибуты$_{opt}$ **get** **;** *атрибуты*$_{opt}$ **set** **;**
  атрибуты$_{opt}$ **set** **;** *атрибуты*$_{opt}$ **get** **;**

Атрибуты, тип и идентификатор в объявлении свойства интерфейса имеют то же значение, что и в объявлении свойства в классе (раздел 10.7).

Коды доступа в объявлении свойства интерфейса соответствуют кодам доступа в объявлении свойства класса (раздел 10.7.2), за исключением того, что телом кода доступа всегда должна быть точка с запятой. Таким образом, коды доступа просто указывают на то, доступно свойство для чтения и записи, только для чтения или только для записи.

Тип свойства интерфейса должен быть безопасным с точки зрения вывода, если присутствует код доступа `get`, и безопасным с точки зрения ввода, если присутствует код доступа `set`.

13.2.3. События интерфейса

События интерфейса объявляются через *объявления-событий-интерфейса*:

объявление-события-интерфейса:
  атрибуты$_{opt}$ **new**$_{opt}$ **event** *тип* *идентификатор* **;**

Атрибуты, тип и *идентификатор* в объявлении события интерфейса имеют то же значение, что и в объявлении события в классе (раздел 10.8).

Тип события интерфейса должен быть безопасным с точки зрения ввода.

13.2.4. Индексаторы интерфейса

Индексаторы интерфейса объявляются через *объявления-индексаторов-интерфейса*:

объявление-индексатора-интерфейса:
  атрибуты$_{opt}$ **new**$_{opt}$ *тип* **this** **[** *список-формальных-параметров* **]**
    { *коды-доступа-интерфейса* **}**

Атрибуты, тип и *список-формальных-параметров* в объявлении индексатора интерфейса имеют то же значение, что и в объявлении индексатора в классе (раздел 10.9).

Коды доступа в объявлении индексатора интерфейса соответствуют кодам доступа в объявлении индексатора в классе (раздел 10.9), за исключением того, что телом кода доступа всегда должна быть точка с запятой. Таким образом, коды доступа просто указывают на то, доступен индексатор для чтения и записи, только для чтения или только для записи.

Все типы формальных параметров в индексаторе интерфейса должны быть безопасными с точки зрения ввода. Помимо этого, любые типы формальных параметров `out` или `ref` также должны быть безопасны с точки зрения вывода. Заметьте, что из-за ограничений базовой платформы выполнения даже параметры `out` должны быть безопасными с точки зрения ввода.

Тип индексатора интерфейса должен быть безопасным с точки зрения вывода, если присутствует код доступа `get`, и безопасным с точки зрения ввода, если присутствует код доступа `set`.

13.2.5. Доступ к элементам интерфейса

Доступ к элементам интерфейса осуществляется с помощью выражений доступа к элементам (раздел 7.6.4) и доступа к индексаторам (раздел 7.6.6.2) в форме `I.M` и `I[A]` соответственно, где `I` — тип интерфейса, `M` — метод, свойство или событие этого типа интерфейса, а `A` — список аргументов индексатора.

Для интерфейсов, которые наследуются строго однократно (каждый интерфейс в цепочке наследований или не имеет базовых интерфейсов, или имеет ровно один), используются точно те же правила поиска элементов (раздел 7.4), вызова методов (раздел 7.6.5.1) и доступа к индексаторам (раздел 7.6.6.2), что для классов и структур: элементы, находящиеся ниже в иерархии наследования, скрывают элементы, находящиеся выше и имеющие такие же имена или сигнатуры. Однако в случае интерфейсов с множественным наследованием могут возникать двусмысленные ситуации, когда два или более несвязанных базовых интерфейса объявляют элементы с одинаковыми именами или сигнатурами. В этом разделе приведены несколько примеров таких ситуаций. Во всех случаях для разрешения двусмысленности можно использовать явное приведение типов.

Пример:

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter : IList, ICounter { }

class C
{
    void Test(IListCounter x)
    {
        x.Count(1);                 // Ошибка
        x.Count = 1;                // Ошибка
        ((IList)x).Count = 1;       // Все в порядке:
                                    // вызывается IList.Count.set
        ((ICounter)x).Count(1);     // Все в порядке:
                                    // вызывается ICounter.Count
    }
}
```

В этом примере два первых оператора приводят к ошибкам компиляции, поскольку поиск элемента (раздел 7.4) `Count` в `IListCounter` является двусмысленным. Как показано в примере, двусмысленность разрешается путем приведения `x` к подходящему типу базового интерфейса. Такие приведения не приводят к накладным расходам во время выполнения — они только указывают на то, что на этапе компиляции экземпляр должен рассматриваться как стоящий выше в иерархии наследования.

Пример:

```
interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber : IInteger, IDouble { }

class C
{
    void Test(INumber n)
    {
        n.Add(1);                    // Вызывается IInteger.Add
        n.Add(1.0);                  // Подходит только IDouble.Add
        ((IInteger)n).Add(1);        // Кандидатом является
                                     // только IInteger.Add
```

```
        ((IDouble)n).Add(1);          // Кандидатом является
                                      // только IDouble.Add
    }
}
```

Здесь при вызове `n.Add(1)` в результате применения правил разрешения перегрузки, описанных в разделе 7.5.3, выбирается `IInteger.Add`. Схожим образом, при вызове `n.Add(1.0)` выбирается `IDouble.Add`. Когда используются явные приведения типов, в качестве кандидата остается только один метод, поэтому двусмысленность отсутствует.

Пример:

```
interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);                 // Вызывается ILeft.F
        ((IBase)d).F(1);        // Вызывается IBase.F
        ((ILeft)d).F(1);        // Вызывается ILeft.F
        ((IRight)d).F(1);       // Вызывается IBase.F
    }
}
```

В этом примере элемент `IBase.F` скрыт элементом `ILeft.F`. Таким образом, при вызове `d.F(1)` выбирается `ILeft.F`, несмотря на то что в пути доступа через `IRight` метод `IBase.F` не является скрытым.

Интуитивно понятное правило сокрытия при множественном наследовании интерфейсов заключается в следующем: если элемент скрыт в любом из путей доступа, он скрыт во всех путях доступа. Так как путь доступа из `IDerived` через `ILeft` в `IBase` скрывает `IBase.F`, этот элемент также скрыт в пути доступа из `IDerived` через `IRight` в `IBase`.

13.3. Полные имена элементов интерфейса

Иногда к элементу интерфейса обращаются через его **полное имя**. Оно состоит из имени интерфейса, в котором объявлен этот элемент, точки и имени элемента.

Полное имя элемента ссылается на интерфейс, в котором объявлен элемент. Рассмотрим следующие объявления:

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}
```

Здесь полным именем `Paint` является `IControl.Paint`, а полным именем `SetText` — `ITextBox.SetText`.

В приведенном примере нельзя ссылаться на `Paint` как на `ITextBox.Paint`.

Когда интерфейс входит в пространство имен, полное имя элемента интерфейса включает в себя имя этого пространства.

Пример:

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

Здесь полным именем метода `Clone` является `System.ICloneable.Clone`.

13.4. Реализации интерфейсов

Интерфейсы могут быть реализованы классами и структурами. Чтобы показать, что класс или структура непосредственно реализует интерфейс, его идентификатор включается в список базовых классов структуры или класса.
Пример:

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry : ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

ДЖОН СКИТ

Меня раздражает, что интерфейсы обычно реализовываются так неявно. Нет никаких подсказок о том, что методы `CompareTo` и `Clone` каким-либо образом связаны с интерфейсами, которые они реализуют. При переопределении методов, унаследованных от базового класса, модификатор `override` делает переопределение очевидным: любой, кто будет проводить рефакторинг класса и захочет переименовать метод, будет знать, что он связан с каким-то другим методом. Для интерфейсов такая подсказка отсутствует.

К сожалению, добавление модификатора в объявления несколько нарушило бы целостность схемы реализации интерфейсов. Код, в котором объявляется метод, может и не знать о том, что он используется для реализации интерфейса:

```
interface IFoo { void Foo(); }
class Base
{
    public void Foo();
}
// IFoo.Foo реализован методом Base.Foo
class Derived : Base, IFoo { }
```

Это, вероятно, самый практичный подход к интерфейсам, но с точки зрения чистоты кода (или перестраховщиков) он выглядит немного неправильным.

Класс или структура, которые непосредственно реализуют интерфейс, также неявным образом непосредственно реализуют все его базовые интерфейсы. Это верно, даже если эти базовые интерфейсы не перечислены явно в списке базовых классов структуры или класса. Пример:

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

class TextBox : ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

Здесь класс `TextBox` реализует как `IControl`, так и `ITextBox`.

Когда класс `C` непосредственно реализует интерфейс, все унаследованные от `C` классы также реализуют этот интерфейс неявным образом. Базовые интерфейсы, указанные в объявлении класса, могут являться сконструированными типами интерфейсов (раздел 4.4). Базовый интерфейс сам по себе не может являться параметром типа, однако он может задействовать параметры типов, находящиеся в области видимости. В следующем коде показано, как класс может реализовывать и расширять сконструированные типы:

```
class C<U, V> { }

interface I1<V> { }

class D : C<string, int>, I1<string> { }

class E<T> : C<int, T>, I1<T> { }
```

Базовые интерфейсы в объявлении обобщенного класса должны удовлетворять правилу уникальности, описанному в разделе 13.4.2.

13.4.1. Явные реализации элементов интерфейса

Для реализации интерфейсов в классе или структуре могут быть объявлены **явные реализации элементов интерфейса**. Явная реализация элемента интерфейса — это объявление метода, свойства, события или индексатора, которое ссылается на полное имя элемента интерфейса.
Пример:

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

Здесь `IDictionary<int, T>.this` и `IDictionary<int, T>.Add` являются явными реализациями элементов интерфейса.

Иногда имя элемента интерфейса может оказаться неподходящим для использования в реализующем его классе. В этом случае элемент интерфейса можно реализовать с помощью явной реализации элемента интерфейса. Например, класс, реализующий файловую абстракцию, скорее всего, будет реализовывать элемент-функцию `Close`, предназначенную для освобождения файлового ресурса, а также метод `Dispose` интерфейса `IDisposable`, используя явную реализацию элемента интерфейса:

```
interface IDisposable
{
    void Dispose();
}

class MyFile : IDisposable
{
    void IDisposable.Dispose()
```

```
    {
        Close();
    }

    public void Close()
    {
        // Произвести действия, необходимые для закрытия файла
        System.GC.SuppressFinalize(this);
    }
}
```

Доступ к явной реализации элемента интерфейса через его полное имя невозможно осуществить при вызове метода, доступе к свойству или индексатору. Она доступна только через экземпляр интерфейса, и в этом случае на нее можно ссылаться просто по имени его элемента.

Использование модификаторов доступа для явной реализации элементов интерфейса приводит к ошибке компиляции. Также к ошибке приводит использование модификаторов `abstract`, `virtual`, `override` и `static`.

Параметры доступа для явных реализаций элементов интерфейса отличаются от остальных элементов. Так как к явным реализациям элементов интерфейса невозможно получить доступ через их полные имена при вызове метода или доступе к свойству, они в некоторой степени являются закрытыми. Однако поскольку они доступны через экземпляр интерфейса, они в некоторой степени также являются и открытыми.

ВЛАДИМИР РЕШЕТНИКОВ

При проверке ограничений на доступность (см. раздел 3.5.4) явные реализации интерфейсов считаются закрытыми.

Явные реализации элементов интерфейса служат двум основным целям:

- Поскольку явные реализации элементов интерфейса недоступны через экземпляры классов или структур, они позволяют исключать реализации интерфейсов из открытого интерфейса класса или структуры. Это особенно полезно, когда класс или структура реализует внутренний интерфейс, который не представляет интереса для пользователя этого класса или структуры.
- Явные реализации элементов интерфейса позволяют разрешать двусмысленность для элементов интерфейса с одинаковыми сигнатурами. Без явных реализаций было бы невозможным предоставлять в классе или структуре различные реализации элементов интерфейса с одинаковыми сигнатурами и типами возвращаемых значений. Также было бы невозможным предоставлять в классе или структуре какую-либо реализацию элементов интерфейса с одинаковыми сигнатурами, но разными типами возвращаемых значений.

ДЖОН СКИТ

В некоторой степени каноническим примером второго случая является `IEnumerable<T>`, который расширяет контракт `IEnumerable`. Это означает, что реализации должны предоставлять два метода:

продолжение ⇨

```
IEnumerator GetEnumerator()
IEnumerator<T> GetEnumerator()
```

Обычно в этом случае необобщенный `IEnumerable.GetEnumerator()` явно вызывает обобщенную версию — такая схема работает, поскольку `IEnumerator<T>` и `IEnumerator` имеют один и тот же тип взаимосвязи.

Чтобы явная реализация элемента интерфейса была корректной, класс или структура должны указывать в списке базовых классов имя интерфейса, содержащего элемент, полное имя, тип и типы параметров которого в точности совпадают с используемыми в явной реализации элемента интерфейса. Рассмотрим следующий класс:

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}       // Некорректно
}
```

Здесь объявление `IComparable.CompareTo` приводит к ошибке компиляции, поскольку `IComparable` не указан в списке базовых классов `Shape` и не является базовым интерфейсом для `ICloneable`. Такая же ситуация наблюдается и в следующих объявлениях:

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}

class Ellipse: Shape
{
    object ICloneable.Clone() {...}       // Некорректно
}
```

Здесь объявление `ICloneable.Clone` в `Ellipse` приводит к ошибке компиляции, поскольку `ICloneable` не указан явно в списке базовых классов `Ellipse`.

Полное имя элемента интерфейса должно ссылаться на интерфейс, в котором был объявлен этот элемент. Пример:

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

В этом примере явная реализация элемента `Paint` должна быть записана как `IControl.Paint`.

13.4.2. Уникальность реализованных интерфейсов

Интерфейсы, которые реализованы объявлением обобщенного типа, должны оставаться уникальными для всех возможных сконструированных типов. Без этого правила было бы невозможно определить, какой метод требуется вызвать для определенных сконструированных типов. Предположим, к примеру, что было бы допустимо записывать объявление обобщенного класса следующим образом:

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>        // Ошибка: конфликт I<U> и I<V>
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Если бы это было допустимо, было бы невозможно определить, какой код должен выполняться в следующей ситуации:

```
I<int> x = new X<int,int>();
x.F();
```

Для определения корректности списка интерфейсов в объявлении обобщенного типа выполняются следующие действия:

- Пусть `L` будет списком интерфейсов, непосредственно указанных в объявлении обобщенного класса, структуры или интерфейса `C`.
- Добавить в `L` любые базовые интерфейсы интерфейсов, уже находящихся в `L`.
- Удалить из `L` все дубликаты.
- Если для какого-либо сконструированного типа, созданного из `C`, после подстановки в `L` аргументов типов может возникнуть ситуация, в которой два интерфейса в `L` будут идентичными, объявление `C` считается некорректным. При определении всех возможных сконструированных типов объявления ограничений не рассматриваются.

В приведенном выше объявлении класса `X` список интерфейсов `L` состоит из `I<U>` и `I<V>`. Объявление является некорректным, так как любой сконструированный тип, в котором `U` и `V` представляют собой один и тот же тип, приведет к ситуации, в которой эти два интерфейса окажутся идентичными.

Допустимо объединять интерфейсы, указанные на разных уровнях иерархии наследования:

```
interface I<T>
{
    void F();
}
```

продолжение ⇨

```
class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V>        // Все в порядке
{
    void I<V>.F() {...}
}
```

Этот код корректен, даже несмотря на то что классом `Derived<U,V>` реализуются как `I<U>`, так и `I<V>`. Код

```
I<int> x = new Derived<int,int>();
x.F();
```

вызывает метод из `Derived`, так как `Derived<int,int>` фактически заново реализует `I<int>` (раздел 13.4.6).

> **ЭРИК ЛИППЕРТ**
> Несмотря на то что недопустимо создавать классы, при конструировании которых два интерфейса могут стать идентичными, обобщенные ковариантность и вариантность в C# 4.0 делают возможным создание классов, порождающих двусмысленность, различными гораздо более хитрыми способами. Представьте, к примеру, базовый класс, реализующий `IEnumerable<object>`, и унаследованный класс, реализующий `IEnumerable<string>`. В C# 3.0 такие типы были несовместимы, но теперь это не так. Поскольку `IEnumerable<string>` теперь можно преобразовать в `IEnumerable<object>`, становится не совсем понятным, какая реализация будет вызвана при вызове метода интерфейса `IEnumerable<Object>` в унаследованном классе. Такие причудливые сценарии склонны демонстрировать поведение среды выполнения, зависящее от реализации, поэтому при любой возможности их следует избегать.

13.4.3. Реализация обобщенных методов

Когда метод интерфейса неявно реализуется обобщенным методом, ограничения на каждый параметр типа в методе должны быть эквивалентны в обоих объявлениях (после замены всех параметров типа в интерфейсе соответствующими аргументами типа), при этом параметры типа в методе рассматриваются по порядку, слева направо.

Однако когда метод интерфейса явно реализуется обобщенным методом, использование ограничений в реализующем методе недопустимо. Вместо этого ограничения наследуются от метода интерфейса:

```
interface I<A, B, C>
{
    void F<T>(T t) where T : A;
    void G<T>(T t) where T : B;
    void H<T>(T t) where T : C;
}
```

```
class C : I<object, C, string>
{
    public void F<T>(T t) {...}                         // Все в порядке
    public void G<T>(T t) where T : C {...}             // Все в порядке
    public void H<T>(T t) where T : string {...}        // Ошибка
}
```

`I<object,C,string>.F<T>` неявно реализован методом `C.F<T>`. В этом случае для `C.F<T>` не требуется (и недопустимо) указывать ограничение `T: object`, так как `object` — это неявное ограничение на все параметры типов. `I<object,C,string>.G<T>` неявно реализован методом `C.G<T>`, так как после замены параметров типа в интерфейсе соответствующими аргументами типа ограничения совпадают с ограничениями в интерфейсе. Ограничение для метода `C.H<T>` является ошибочным, поскольку бесплодные типы (в данном случае `string`) не могут использоваться в качестве ограничений. При отсутствии ограничения все равно бы возникала ошибка, поскольку ограничения для неявных реализаций методов интерфейса должны совпадать. Таким образом, невозможно неявно реализовать `I<object,C,string>`. Этот метод интерфейса должен реализовываться только с использованием явной реализации элемента интерфейса:

```
class C: I<object,C,string>
{
    ...

    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;        // Все в порядке
        H<T>(t);
    }
}
```

В этом примере явная реализация элемента интерфейса вызывает открытый метод, имеющий строго более слабые ограничения. Заметьте, что присваивание `s = t` корректно, так как `T` наследует ограничение `T: string`, несмотря на то что это ограничение невозможно выразить в исходном коде.

13.4.4. Сопоставление интерфейсов

Класс или структура должны предоставлять реализации для всех элементов интерфейсов, перечисленных в списке базовых классов. Процесс поиска реализаций элементов интерфейсов в реализующих их классе или структуре называется **сопоставлением интерфейсов (interface mapping)**.

В процессе сопоставления интерфейсов для класса или структуры `C` производится поиск реализации каждого элемента каждого интерфейса, перечисленного в списке базовых классов `C`. Реализация конкретного элемента интерфейса `I.M`, где `I` — интерфейс, в котором объявлен элемент `M`, определяется в процессе проверки каждого класса или структуры `S`, начиная с `C` и повторяя поиск последовательно во всех базовых классах `C`, пока не будет найдено соответствие:

- Если `S` содержит объявление явной реализации элемента интерфейса, которая соответствует `I` и `M`, этот элемент является реализацией `I.M`.
- В противном случае, если `S` содержит объявление нестатического открытого элемента, соответствующего `M`, этот элемент является реализацией `I.M`. Если соответствует более чем один элемент, то не определено, какой из них является реализацией `I.M`. Эта ситуация может возникнуть, только если `S` — сконструированный тип, в котором два элемента, согласно объявлению в обобщенном типе, имеют разные сигнатуры, однако аргументы типа делают их сигнатуры идентичными.

Если реализации не могут быть найдены для всех элементов всех интерфейсов из списка базовых классов `C`, возникает ошибка компиляции. Заметьте, что к элементам интерфейса также относятся элементы, унаследованные от базовых интерфейсов.

При сопоставлении интерфейсов элемент `A` какого-либо класса соответствует элементу `B` интерфейса, если:

- `A` и `B` — методы, и их имена, типы и списки формальных параметров идентичны.
- `A` и `B` — свойства, их имена и типы идентичны, и `A` имеет такие же коды доступа, что и `B` (допустимо наличие у `A` дополнительных кодов досупа, если `A` не является явной реализацией элемента интерфейса).
- `A` и `B` — события, и их имена и типы идентичны.
- `A` и `B` — индексаторы, типы и списки формальных параметров `A` и `B` идентичны, и `A` имеет такие же коды доступа, что и `B` (допустимо наличие у `A` дополнительных кодов доступа, если `A` не является явной реализацией элемента интерфейса).

Из алгоритма сопоставления интерфейсов можно сделать следующие важные выводы:

- При определении элемента класса или структуры, реализующего элемент интерфейса, явные реализации элементов интерфейса имеют приоритет перед остальными элементами того же класса или структуры.
- Статические элементы, а также элементы, не являющиеся открытыми, не участвуют в сопоставлении интерфейсов.

Пример:

```
interface ICloneable
{
    object Clone();
}

class C : ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

Здесь реализацией `Clone` из `ICloneable` становится элемент `ICloneable.Clone` из класса `C`, поскольку явные реализации элементов интерфейса имеют приоритет перед остальными элементами.

Если класс или структура реализует два или более интерфейсов, содержащих элементы с одинаковыми именами, типами и типами параметров, можно сопоставить каждый из этих элементов интерфейса одному элементу класса или структуры. Пример:

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page : IControl, IForm
{
    public void Paint() {...}
}
```

В этом примере оба метода `Paint` из `IControl` и `IForm` сопоставляются методу `Paint` из `Page`. Разумеется, можно также создать отдельные явные реализации для каждого из методов.

Если класс или структура реализует интерфейс, содержащий скрытые элементы, некоторые из элементов обязательно должны быть реализованы с использованием явных реализаций элементов интерфейса. Пример:

```
interface IBase
{
    int P { get; }
}

interface IDerived : IBase
{
    new int P();
}
```

Реализация этого интерфейса потребует хотя бы одну явную реализацию элемента и будет иметь одну из следующих форм:

```
class C : IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}

class C : IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}

class C : IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}
```

Когда класс реализует несколько интерфейсов, имеющих один и тот же базовый интерфейс, в этом классе может присутствовать только одна реализация базового интерфейса.

Пример:

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

class ComboBox : IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
    void IListBox.SetItems(string[] items) {...}
}
```

В этом примере невозможно создать отдельные реализации для `IControl`, перечисленного в списке базовых классов, `IControl`, унаследованного интерфейсом `ITextBox`, и `IControl`, унаследованного интерфейсом `IListBox`. Более того, здесь трудно представить различия между этими интерфейсами. Наоборот, реализации `ITextBox` и `IListBox` разделяют одну и ту же реализацию `IControl`, а `ComboBox` просто рассматривается как класс, реализующий три интерфейса: `IControl`, `ITextBox` и `IListBox`.

Элементы базового класса участвуют в сопоставлении интерфейсов.

Пример:

```
interface Interface1
{
    void F();
}

class Class1
{
    public void F() { }
    public void G() { }
}

class Class2 : Class1, Interface1
{
    new public void G() { }
}
```

Здесь метод `F` класса `Class1` используется в реализации интерфейса `Interface1` классом `Class2`.

13.4.5. Наследование реализаций интерфейсов

Класс наследует все реализации интерфейсов, предоставляемые его базовыми классами.

> **БИЛЛ ВАГНЕР**
>
> Один из предыдущих комментариев Эрика насчет того, что слово «наследование» выбрано неудачно, для этого раздела подходит даже в большей степени. Поведение методов интерфейсов отличается от поведения как виртуальных, так и невиртуальных методов, объявленных в базовых классах. Может потребоваться некоторое время, чтобы научиться понимать, какой из методов является наилучшим выбором, когда несколько классов в иерархии наследования объявляют реализации интерфейса. В процессе чтения этого раздела вы встретите множество различных правил для выбора наилучшего метода в случае, когда этот метод определен в интерфейсе.

Без явной **повторной реализации** интерфейса наследующий класс никаким способом не может изменить сопоставления интерфейсов, которые он унаследовал от своих базовых классов. Рассмотрим, к примеру, объявления:

```
interface IControl
{
    void Paint();
}

class Control : IControl
{
    public void Paint() {...}
}

class TextBox : Control
{
    new public void Paint() {...}
}
```

Здесь метод `Paint` в классе `TextBox` скрывает метод `Paint` из класса `Control`, но не изменяет сопоставление `Control.Paint` методу `IControl.Paint`, поэтому вызовы `Paint` через экземпляры классов и экземпляры интерфейсов приведут к следующим результатам:

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();                      // Вызывает Control.Paint();
t.Paint();                      // Вызывает TextBox.Paint();
ic.Paint();                     // Вызывает Control.Paint();
it.Paint();                     // Вызывает Control.Paint();
```

Однако когда метод интерфейса сопоставляется виртуальному методу класса, унаследованные классы могут переопределять этот виртуальный метод и изменять реализацию интерфейса. Например, приведенные выше объявления можно переписать следующим образом:

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public virtual void Paint() {...}
}

class TextBox: Control
{
    public override void Paint() {...}
}
```

В этом случае результаты будут следующими:

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();                    // Вызывает Control.Paint();
t.Paint();                    // Вызывает TextBox.Paint();
ic.Paint();                    // Вызывает Control.Paint();
it.Paint();                    // Вызывает TextBox.Paint();
```

Так как явные реализации элементов интерфейса не могут быть объявлены с модификатором `virtual`, переопределять их невозможно. Однако из явной реализации можно вызывать другой метод, а он может быть объявлен как `virtual`, тем самым позволяя унаследованным классам переопределять его.

Пример:

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Здесь классы, унаследованные от `Control`, могут уточнить реализацию `IControl.Paint`, переопределив метод `PaintControl`.

13.4.6. Повторная реализация интерфейса

Класс, наследующий реализацию интерфейса, может **повторно реализовать** интерфейс, включив его в список базовых классов.

Повторная реализация интерфейса подчиняется тем же самым правилам сопоставления интерфейсов, что и первоначальная. Таким образом, сопоставление унаследованного интерфейса никоим образом не влияет на сопоставление для повторной реализации.

Пример:

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}
```

Тот факт, что класс `Control` сопоставляет метод `IControl.Paint` методу `Control.IControl.Paint`, не влияет на повторную реализацию в `MyControl`, сопоставляющую метод `IControl.Paint` методу `MyControl.Paint`.

Унаследованные объявления открытых элементов и унаследованные явные объявления элементов интерфейса участвуют в процессе сопоставления для повторно реализованных интерфейсов. Пример:

```
interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}
```

Здесь реализация `IMethods` в классе `Derived` сопоставляет методы интерфейса методам `Derived.F`, `Base.IMethods.G`, `Derived.IMethods.H` и `Base.I`.

Когда класс осуществляет реализацию интерфейса, он также неявно реализует все базовые интерфейсы этого интерфейса. Аналогично, повторная реализация интерфейса также неявно повторно реализует все его базовые интерфейсы. Пример:

```
interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}
```

Здесь повторная реализация `IDerived` также повторно реализует `IBase`, сопоставляя метод `IBase.F` методу `D.F`.

> **ДЖОН СКИТ**
>
> Хотя такое поведение в некоторых ситуациях действительно полезно, его обычно следует избегать, потому что оно является источником огромной путаницы. Как правило, каждый вызов, использующий одно и то же имя метода и один и тот же список аргументов для одного и того же объекта, должен приводить к вызову одного и того же метода. Существуют разные способы выйти из этого замечательного состояния, и все они должны использоваться с чрезвычайной осторожностью и только там, где это на самом деле необходимо.

13.4.7. Абстрактные классы и интерфейсы

Абстрактный класс, так же как и неабстрактный, должен предоставлять реализации всех методов интерфейсов, перечисленных в его списке базовых классов. Однако абстрактный класс может сопоставлять методы интерфейса абстрактным методам. Пример:

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

Здесь реализация `IMethods` сопоставляет методы `F` и `G` абстрактным методам, которые должны быть переопределены в неабстрактных классах, наследующихся от `C`.

Заметьте, что явные реализации элементов интерфейса не могут быть абстрактными, но они, разумеется, могут вызывать абстрактные методы. Пример:

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

Здесь в неабстрактных классах, наследующихся от `C`, необходимо будет переопределить методы `FF` и `GG`, тем самым предоставив фактическую реализацию интерфейса `IMethods`.

Глава 14

Перечисления

Перечисление — отдельный тип-значение (раздел 4.1), содержащий совокупность именованных констант.

> **ДЖОН СКИТ**
>
> Примечательно, что эти «именованные константы» на самом деле просто числа. Они не могут выражать поведение, в отличие от почти всех остальных средств C#. Это одна из *крайне* немногих областей, где язык Java более выразителен, чем C#. В Java перечисления являются гораздо более мощными: в перечислении можно объявлять методы, а затем переопределять их, например, для некоторых значений. Несмотря на то что некоторые особенности Java-перечислений можно эмулировать в C#, языковая поддержка (и поддержка в каркасе) в будущей версии будет очень приветствоваться.

Пример:

```
enum Color
{
   Red,
   Green,
   Blue
}
```

Здесь объявлено перечисление с именем `Color`, включающее элементы `Red`, `Green` и `Blue`.

14.1. Объявление перечисления

Объявление перечисления задает новый перечислимый тип. Объявление перечисления начинается с ключевого слова `enum` и задает имя, базовый тип и элементы перечисления.

объявление-перечисления:
  атрибуты$_{opt}$ *модификаторы*$_{opt}$ **enum** *имя* *базовый-тип*$_{opt}$ *тело-перечисления* **;**$_{opt}$

базовый-тип:
  : *целочисленный-тип*

тело-перечисления:
  { *объявление-элементов-перечисления*$_{opt}$ **}**
  { *объявление-элементов-перечисления*$_{opt}$ **,** **}**

У каждого перечисления есть соответствующий целочисленный тип, который называют *базовым типом*. Этот базовый тип должен позволять представлять все перечисляемые значения, объявленные в перечислении. В перечислении можно явным образом объявлять базовые типы `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`.

> **ДЖОН СКИТ**
>
> Это одно из немногих мест в языке, где вы не можете заменить «сокращенную» версию типа с его полным эквивалентом. Например, вы не можете объявить перечисление `Color` следующим образом:
>
> `enum Color: System.Int64 { ... }`

Пример:

```
enum Color : long
{
   Red,
   Green,
   Blue
}
```

Здесь представлено объявление перечисления с базовым типом `long`. Разработчик может выбрать базовый тип `long`, как показано в примере, с той целью, чтобы использовать диапазон значений `long`, а не диапазон значений `int`, или оставить этот параметр на будущее.

14.2. Модификаторы перечисления

В *объявлении-перечисления* может стоять необязательная последовательность модификаторов перечисления:

```
модификаторы-перечисления:
   модификатор-перечисления
   модификаторы-перечисления   модификатор-перечисления

модификатор-перечисления:
   new
   public
   protected
   internal
   private
```

Компилятор выдаст ошибку, если в объявлении перечисления указано несколько модификаторов.

Модификаторы объявления перечисления имеют такое же значение, как и для объявления классов (раздел 10.1.1). Замечание: в объявлении перечисления не допускается использование модификаторов `abstract` и `sealed`. Перечисления не могут быть абстрактными и не могут служить базовыми классами при наследовании.

14.3. Элементы перечисления

Тело перечисления определяет ноль или более элементов перечисления, являющихся именованными константами перечисления. Два элемента не могут иметь одно и то же имя.

```
объявление-элементов-перечисления:
   объявление-элемента-перечисления
   объявление-элемента-перечисления   ,   объявление-элемента-перечисления

объявление-элементов-перечисления:
   атрибутыopt   идентификатор
   атрибутыopt   идентификатор   =   константное-выражение
```

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> В реализации Microsoft C# элемент перечисления не может носить имя «value__», потому что это имя зарезервировано для внутренних представлений перечислений.

Каждый элемент перечисления имеет связанное с ним константное значение. Тип этого значения определяется базовым типом, который указывается в объявлении перечисления. Значение константы для каждого элемента должно находиться в диапазоне значений базового типа этого перечисления. Например:

```
enum Color : uint
{
   Red = -1,
   Green = -2,
   Blue = -3
}
```

в результате компиляции будет выдана ошибка, потому что константные значения `-1`, `-2` и `-3` не находятся в диапазоне базового типа `uint`.

Несколько элементов перечисления могут иметь одно и то же связанное с ним значение. Например:

```
enum Color
{
   Red,
   Green,
   Blue,

   Max = Blue
}
```

В этом перечислении два элемента — `Blue` и `Max` — имеют одно и то же связанное с ними значение.

Связанное значение элементов перечисления задается либо явно, либо неявно. Если при объявлении элемента перечисления указан инициализатор в виде *константного-выражения*, значение этого константного выражения, неявно преобразованное в базовый тип перечисления, является связанным значением элемента перечисления. Если в объявлении элемента перечисления инициализатор не задан, то связанное значение устанавливается неявно, а именно:

- Первый элемент перечисления автоматически принимает значение `0`.
- Последующие элементы перечисления принимают значение предыдущего элемента, увеличенного на `1`. Увеличенное значение должно находиться в диапазоне значений базового типа, в противном случае выдается ошибка компиляции.

ДЖОН СКИТ

Использование значений по умолчанию почти всегда является неверным там, где нужно использовать побитовые операции. Применение атрибута `[Flags]` в объявлении перечисления позволяет работать с перечислением как с набором битов, где значениями элементов обычно являются 1, 2, 4, 8 и т. д., а элемент «None» имеет значение `0`. Язык мог бы помочь разработчикам избавиться от случайного использования неверных значений, но без добавления большей сложности, чем оно того стоит, тяжело судить, насколько хорошо эта цель может быть достигнута.

БИЛЛ ВАГНЕР

Я бы предпочел увидеть следующий пример, реализованный с использованием методов расширения. «`Color.Red.StringFromColor()`» для меня читается лучше, чем просто «`StringFromColor(Color.Red)`».
Здесь хочется сослаться на комментарий Джона о неспособности адекватно выразить поведение.

Пример:

```
using System;
enum Color
{
   Red,
   Green = 10,
   Blue
}
class Test
{
   static void Main()
   {
      Console.WriteLine(StringFromColor(Color.Red));
      Console.WriteLine(StringFromColor(Color.Green));
      Console.WriteLine(StringFromColor(Color.Blue));
   }
   static string StringFromColor(Color c)
   {
      switch (c)
      {
         case Color.Red:
         return String.Format("Red = {0}", (int)c);
         case Color.Green:
         return String.Format("Green = {0}", (int)c);
         case Color.Blue:
         return String.Format("Blue = {0}", (int)c);
```

продолжение ➩

```
            default:
            return "Invalid color";
        }
    }
}
```

Этот код выведет имена элементов перечисления и их соответствующие значения. В результате получим

```
Red = 0
Green = 10
Blue = 11
```

по следующим причинам:

- Элементу перечисления `Red` автоматически присваивается значение нуль (так как его значение не было инициализировано и так как этот элемент располагается первым в перечислении).
- Элементу `Green` явно задается значение `10`.
- Элементу `Blue` автоматически присваивается значение, на единицу большее, чем расположенному перед ним элементу.

Значение, связанное с элементом перечисления, не может прямо или косвенно использовать собственное значение. Помимо этого ограничения зацикливания, элементы перечисления могут свободно ссылаться на инициализаторы других элементов перечисления независимо от их расположения.

Например:

```
enum Circular
{
    A = B,
    B
}
```

результатом компиляции будет ошибка, поскольку объявления `A` и `B` образуют циклическую зависимость. Элемент перечисления `A` имеет явную зависимость от элемента `B`, в то время как значение элемента `B` зависит от значения элемента `A`.

Имена и области действия элементов перечислений полностью аналогичны таковым для полей классов. Областью действия элемента перечисления является тело перечислимого типа, содержащего этот элемент. Внутри этой области действия к элементам перечисления можно обращаться с помощью простых имен. Из всего остального кода имя элемента перечисления необходимо предварять именем его перечислимого типа. Для элементов перечисления не задается доступность —элемент перечисления доступен, если доступно содержащее его перечисление.

14.4. Тип System.Enum

`System.Enum` представляет собой абстрактный базовый класс для всех перечислений (это не то, что называется базовым типом перечисления), и элементы, унаследованные от `System.Enum`, доступны в любом перечислимом типе. Существует преобразование упаковки (раздел 4.3.1) из любого перечисления в тип `System.`

`Enum` и преобразование распаковки (раздел 4.3.2) из `System.Enum` в любое перечисление.

Заметьте, что сам тип `System.Enum` не является *перечислением*. Напротив, это *класс*, который является базовым для всех *перечислений*. Тип `System.Enum` наследуется от `System.ValueType` (раздел 4.1.1), который, в свою очередь, наследуется от `object`. На стадии выполнения значение типа `System.Enum` может быть `nul` или ссылкой на упакованное значение любого перечислимого типа.

14.5. Значения перечислений и операции

Каждое перечисление определяет отдельный тип; для преобразования между перечислением и целым типом или между двумя перечислениями требуется явное приведение (раздел 6.2.2). Элементы, которые может включать в себя перечисление, не ограничен содержащимися в нем элементами. В частности, любое значение базового типа может быть приведено к перечислению и является отдельным допустимым значением этого перечисления.

Элементы перечисления имеют тип содержащего их перечисления (кроме тех, которые находятся внутри других инициализаторов элементов перечисления; см. раздел 14.3). Значение элемента перечисления, объявленного в перечислении `E`, связанного со значением `v`, соответствует `(E)v`.

Следующие операции могут использоваться для значений перечислений:
`==`, `!=`, `<`, `>`, `<=`, `>=` (раздел 7.10.5), бинарный `+` (раздел 7.8.4), бинарный `-` (раздел 7.8.5), `^`, `&`, `|` (раздел 7.11.2), `~` (раздел 7.7.4), `++` и `--` (разделы 7.6.9 и 7.7.5).

ДЖОН СКИТ

Некоторые из этих операций имеют реальный смысл только для перечислений, в которых указан атрибут `[Flags]`, но язык не принуждает использовать именно такую конструкцию.

Каждое перечисление является производным от класса `System.Enum` (который, в свою очередь, является производным от `System.ValueType` и `object`). Унаследованные методы и свойства этого класса можно применять для значений перечислений.

Глава 15

Делегаты

Делегаты предназначены для случаев, в которых C++, Pascal и Modula используют указатели на функции. В отличие от последних в C++ делегаты являются полностью объектно-ориентированными. Также, в отличие от используемых в C++ указателей на элементы-функции, делегаты инкапсулируют как экземпляр объекта, так и метод.

> **БИЛЛ ВАГНЕР**
>
> Я испытываю странное чувство изумления, читая эту маленькую главу спецификации языка. Делегаты являются частью C#, начиная с версии 1.0. В то время большинство разработчиков из сообщества C# (включая меня) рассматривали делегаты как некую формальность, связанную с событиями. По прошествии десяти лет я уже не могу представить C# без делегатов. Они каждый день используются большинством из нас в LINQ-запросах, композиции функций, замыканиях и т. д. Делегаты и концепция работы с кодом как с данными являются столь неотъемлемой частью современной экосистемы .NET, что я не могу представить программирование на языке, который не позволяет мне выражать концепцию функций (и действий) как данные.

Объявление делегата определяет класс, унаследованный от класса `System.Delegate`. Экземпляр делегата инкапсулирует список вызова, состоящий из одного или нескольких методов, на каждый из которых ссылаются как на вызываемую сущность. Для методов экземпляра вызываемая сущность включает в себя экземпляр и метод этого экземпляра. Для статических методов она состоит только из метода. Вызов экземпляра делегата с подходящим набором аргументов приводит к вызову каждой из вызываемых сущностей делегата с передачей ей заданного набора аргументов.

Интересным и полезным свойством экземпляра делегата является то, что он не знает и не заботится о классах инкапсулируемых им методов; значение имеет только совместимость этих методов (раздел 15.1) с типом делегата. Это свойство делает делегаты идеальными для «анонимного» вызова.

> **ЭРИК ЛИППЕРТ**
>
> При первом знакомстве делегаты нередко сбивают разработчика с толку. Я предпочитаю думать о делегатах как о чем-то похожем на интерфейс с единственным методом; экземпляр типа делегата в таком случае является объектом, реализующим этот метод.

15.1. Объявления делегатов

Объявление-делегата — это *объявление-типа* (раздел 9.6), которое объявляет новый тип делегата.

```
объявление-делегата:
   атрибуты_opt  модификаторы-делегата_opt  delegate  тип-возвращаемого-значения
      идентификатор список-параметров-вариантного-типа_opt
         (  список-формальных-параметров_opt  )  ограничения-на-параметры-типа_opt  ;

модификаторы-делегата:
   модификаторы-делегата
   модификаторы-делегата   модификатор-делегата

модификатор-делегата:
   new
   public
   protected
   internal
   private
```

Если один и тот же модификатор встречается в объявлении делегата несколько раз, возникает ошибка компиляции.

Модификатор `new` допускается только для делегатов, объявляемых внутри объявления другого типа. В этом случае модификатор указывает, что делегат скрывает унаследованный элемент с таким же именем, как описано в разделе 10.3.4.

Модификаторы `public`, `protected`, `internal` и `private` управляют доступом к делегату. В зависимости от контекста, в котором объявляется делегат, некоторые из этих модификаторов могут быть недопустимы (раздел 3.5.1).

Идентификатор определяет имя делегата.

Необязательный *список-формальных-параметров* определяет параметры делегата, а *тип-возвращаемого-значения* — тип возвращаемого значения делегата.

Необязательный *список-параметров-вариантного-типа* (раздел 13.1.3) определяет список параметров типа для самого делегата.

Тип возвращаемого значения типа делегата должен быть или `void`, или безопасным с точки зрения вывода (output-safe, раздел 13.1.3.1).

Все типы формальных параметров типа делегата должны быть безопасны с точки зрения ввода (input-safe). Помимо этого, любые типы параметров `out` или `ref` должны быть безопасны с точки зрения вывода. Заметьте, что даже параметры `out` должны быть безопасны с точки зрения ввода в силу ограничений базовой среды выполнения.

> **ЭРИК ЛИППЕРТ**
>
> Упомянутое здесь ограничение связано с тем, что на самом деле параметры `out` реализованы как параметры `ref`: вы можете и считывать, и записывать значения параметров `out`. Компилятор требует, чтобы в параметр `out` было записано значение, прежде чем его можно будет прочитать, однако это правило языка C#, а не среды выполнения. Если бы параметры `out` действительно предназначались «только для записи» и это
>
> *продолжение* ⇨

> ограничение присутствовало бы в среде выполнения, их теоретически можно было бы сделать ковариантными. Помните об этом, когда в следующий раз будете разрабатывать новую систему типов.

Эквивалентность типов делегатов в C# определяется именем, а не структурой. А именно, два различных типа делегатов, имеющих одинаковые списки параметров и тип возвращаемого значения, считаются разными типами делегатов. Однако экземпляры двух отдельных, но структурно эквивалентных типов делегатов при сравнении могут рассматриваться как равные (раздел 7.9.8).

Например:

```
delegate int D1(int i, double d);

class A
{
   public static int M1(int a, double b) {...}
}
class B

{
   delegate int D2(int c, double d);
   public static int M1(int f, double g) {...}
   public static void M2(int k, double l) {...}
   public static int M3(int g) {...}
   public static void M4(int g) {...}
}
```

Оба типа делегатов `D1` и `D2` совместимы с методами `A.M1` и `B.M1`, так как имеют тот же тип возвращаемого значения и список параметров; однако эти типы делегатов являются двумя разными типами, поэтому они не взаимозаменяемы. Типы делегатов `D1` и `D2` несовместимы с методами `B.M2`, `B.M3` и `B.M4`, так как имеют разные типы возвращаемого значения и списки параметров.

> **ЭРИК ЛИППЕРТ**
>
> Когда вы проектируете новую систему типов, вы не знаете точно, как она будет использоваться в будущем. Кто-то может представить сценарии использования, включающие различные категории делегатов (такие как «делегат для метода с наблюдаемыми побочными эффектами» или «делегат для метода, который может вызываться параллельно»). В этом случае имело бы смысл запретить присваивания между этими категориями. На практике оказывается, что структурная типизация делегатов очень полезна; на самом деле не существует таких семантических различий между `Predicate<int>` и `Func<int, bool>`, которые должны навязываться средой исполнения. Если бы нам пришлось разрабатывать все сначала, типы делегатов могли бы быть более структурно совместимы, чем они есть сегодня.

Как и в объявлениях других обобщенных типов, для создания сконструированного типа делегата должны быть заданы аргументы типа. Типы параметров и тип возвращаемого значения сконструированного типа делегата создаются путем

замены каждого параметра типа в объявлении делегата соответствующим ему аргументом типа сконструированного типа делегата. Получившиеся в результате тип возвращаемого значения и типы параметров используются для определения того, какие методы совместимы со сконструированным типом делегата. Например:

```
delegate bool Predicate<T>(T value);

class X
{
   static bool F(int i) {...}
   static bool G(string s) {...}
}
```

Тип делегата `Predicate<int>` совместим с методом `X.F`, а тип делегата `Predicate<string>` совместим с методом `X.G`.

Единственным способом объявить делегат является *объявление-делегата*. Делегат — это класс, унаследованный от `System.Delegate`. Из-за наличия неявного модификатора `sealed` не допускается наследование каких-либо типов от типа делегата. Также недопустимо наследовать от `System.Delegate` классы, не являющиеся делегатами. Обратите внимание, что `System.Delegate` сам по себе не является делегатом; это класс, от которого унаследованы все делегаты.

В C# определен специальный синтаксис для создания экземпляра и вызова делегата. За исключением создания экземпляра, любая операция, применимая к классу или экземпляру класса, также может быть применена соответственно к классу или экземпляру делегата. В частности, можно получать доступ к элементам типа `System.Delegate`, используя обычный синтаксис доступа к элементу.

Набор методов, инкапсулированных в экземпляре делегата, называется списком вызова. При создании экземпляра делегата (раздел 15.2) из единственного метода он инкапсулирует этот метод, и его список вызова содержит только одну запись. Однако когда объединяются два ненулевых экземпляра делегата, их списки вызова объединяются в порядке следования операндов — сначала левый, затем правый — для формирования нового списка, содержащего две или более записей.

ДЖОН СКИТ

Эта двойственность одноадресности и многоадресности в некоторых случаях крайне неудобна, а в других — невероятно полезна. Например, имеет смысл говорить о «цели» одноадресного делегата, но в то же время каждая запись в списке вызова многоадресного делегата может иметь отдельную цель. На практике это редко является проблемой, однако может затруднить простое и точное описание делегатов.

Делегаты объединяются с помощью бинарных операций `+` (раздел 7.8.4) и `+=` (раздел 7.17.2). Делегат можно удалить из объединения делегатов с помощью бинарных операций `-` (раздел 7.8.5) и `-=` (раздел 7.17.2). Делегаты также можно проверять на равенство (раздел 7.10.8).

Следующий пример демонстрирует создание нескольких экземпляров делегата и соответствующие им списки вызова:

```
delegate void D(int x);
class C
{
   public static void M1(int i) {...}
   public static void M2(int i) {...}
}

class Test
{
   static void Main()
   {
      D cd1 = new D(C.M1);      // M1
      D cd2 = new D(C.M2);      // M2
      D cd3 = cd1 + cd2;      // M1 + M2
      D cd4 = cd3 + cd1;      // M1 + M2 + M1
      D cd5 = cd4 + cd3;      // M1 + M2 + M1 + M1 + M2
   }
}
```

При создании экземпляров с именами `cd1` и `cd2` каждый из них инкапсулирует один метод. Когда создается `cd3`, его список вызова содержит два метода, `M1` и `M2`, именно в таком порядке. Список вызова `cd4` содержит `M1`, `M2` и `M1` в этом порядке. И наконец, список вызова `cd5` содержит `M1`, `M2`, `M1`, `M1` и `M2`, опять же в таком порядке. Другие примеры объединения (а также удаления) делегатов приведены в разделе 15.4.

15.2. Совместимость делегатов

Метод или делегат `M` *совместим* с делегатом типа `D`, если верны все следующие утверждения:

- `D` и `M` имеют одинаковое число параметров, и каждый параметр `D` имеет такой же модификатор `ref` или `out`, что и соответствующий ему параметр `M`.
- Для каждого параметра-значения (то есть не имеющего модификатора `ref` или `out`) существует тождественное преобразование (раздел 6.1.1) или неявное ссылочное преобразование (раздел 6.1.6) из типа параметра `D` в тип соответствующего параметра `M`.
- Для каждого параметра `ref` или `out` параметр `D` имеет такой же тип, что и `M`.
- Существует тождественное преобразование или неявное преобразование ссылки из типа возвращаемого значения `M` в тип возвращаемого значения `D`.

15.3. Создание экземпляра делегата

Экземпляр делегата создается *выражением-создания-делегата* (раздел 7.6.10.5) или преобразованием в тип делегата. Созданный экземпляр делегата ссылается на одну из следующих сущностей:

- статический метод, указанный в *выражении-создания-делегата*;
- целевой объект (который не может иметь значение `null`) и метод экземпляра, указанные в *выражении-создания-делегата*;
- другой делегат.

  Например:

```
delegate void D(int x);

class C
{
   public static void M1(int i) {...}
   public void M2(int i) {...}
}

class Test
{
   static void Main()
   {
      D cd1 = new D(C.M1);      // Статический метод
      C t = new C();
      D cd2 = new D(t.M2);      // Метод экземпляра
      D cd3 = new D(cd2);       // Другой делегат
   }
}
```

После создания экземпляры делегатов всегда ссылаются на один и тот же целевой объект и метод. Помните, что при объединении двух делегатов или удалении одного из другого получившийся делегат содержит собственный список вызова; списки вызова объединяемых или удаляемых делегатов остаются неизменными.

15.4. Вызов делегата

Для вызова делегата в C# существует специальный синтаксис. Когда вызывается ненулевой экземпляр делегата, список вызова которого содержит одну запись, он вызывает один метод с теми же аргументами, которые были ему переданы, и возвращает то же значение, что и этот метод. (См. раздел 7.6.5.3 для более подробной информации о вызове делегата.) Если в процессе вызова такого делегата возникает исключение и оно не перехватывается внутри вызываемого метода, поиск оператора `catch` продолжается в вызвавшем делегат методе, как если бы он непосредственно вызывал метод, на который ссылается делегат.

Вызов экземпляра делегата, список вызова которого содержит несколько записей, осуществляется путем синхронного вызова каждого метода из списка по порядку. Каждому вызываемому методу передается тот же набор аргументов, который был передан экземпляру делегата. Если при вызове такого делегата используются параметры-ссылки (раздел 10.6.1.2), каждому методу передается ссылка на одну и ту же переменную; изменения этой переменной, сделанные одним методом из списка вызова, будут видимы всем последующим методам. Если при вызове деле-

гата используются выходные параметры или возвращаемое значение, их итоговое значение будет зависеть от вызова последнего делегата в списке.

Если в процессе обработки вызова такого делегата возникает исключение и оно не перехватывается внутри вызываемого метода, поиск оператора `catch` продолжается в методе, вызвавшем делегат, а все остальные методы из списка вызова не вызываются.

> **БИЛЛ ВАГНЕР**
>
> По причине такого поведения, в общем случае вы должны стремиться создавать такие методы, реализующие делегаты, которые ни при каких обстоятельствах не выбрасывают исключений. Это может привести к ошибкам, после которых, скорее всего, будет невозможно надежно восстановить нормальное состояние программы. Данный фактор не так важен, если вы знаете, что ваш делегат будет использоваться только как одноадресный.

> **КРИСТИАН НЕЙГЕЛ**
>
> Существует способ решить проблему с исключениями, выбрасываемыми в методах-обработчиках, на которые ссылается делегат. Вместо того чтобы напрямую вызывать экземпляр делегата, можно воспользоваться его методом `GetInvocationList`, чтобы вызвать каждый делегат из списка вызова по отдельности. Каждый такой вызов можно обезопасить, поместив его в блок `try-catch`. Одним из вариантов действий при возникновении исключения может быть удаление вызвавшего его метода-обработчика из списка вызова.

Попытка вызвать экземпляр делегата, имеющий значение `null`, приводит к возникновению исключения типа `System.NullReferenceException`.

> **ДЖОН СКИТ**
>
> В некоторой степени такой результат полностью обоснован; с другой стороны, он выглядит нелогичным, поскольку нулевое значение ссылки является нормальным представлением пустого списка вызова. Этой путаницы, вероятно, можно было бы избежать, если бы у каждого типа делегата был статический метод `Invoke` — возможно, даже метод расширения, — который выполнял бы соответствующую проверку на нулевое значение. Данное решение было бы более элегантным, если бы компилятор C# автоматически осуществлял эту проверку в каждом вызывающем выражении. Честно говоря, все усложняется, когда тип делегата имеет отличный от `void` тип возвращаемого значения или использует параметр `out`.
>
> Каким бы ни было лучшее решение, необходимость проверки на нулевое значение при каждом вызове делегата, безусловно, раздражает.

> **ЭРИК ЛИППЕРТ**
>
> Распространенным шаблоном кода для «потокобезопасных» делегатов является что-то вроде этого:
>
> ```
> var temp = this.mydelegate;
> if (temp != null) temp();
> ```

вместо более очевидного решения:

```
if (this.mydelegate != null) this.mydelegate();
```

Первый фрагмент безопаснее, так как если существует возможность изменения `mydelegate` в другом потоке, его значение может поменяться на `null` между проверкой и вызовом. Кроме того, такой подход позволяет избежать только одного состояния гонки. Предположим, что мы используем первый вариант кода. Временная переменная сохраняет текущее значение `mydelegate`. В другом потоке `mydelegate` присваивается значение `null`, и глобальное состояние, требующееся предыдущему содержимому `mydelegate` для успешного выполнения, уничтожается. Но именно предыдущее содержимое мы собираемся вызвать! Если ваши делегаты (в частности, делегаты, ассоциированные с событиями) восприимчивы к этой проблеме, вы должны реализовать какой-то другой механизм, позволяющий удостовериться, что при возникновении гонки не произойдет ничего плохого. Возможно, наилучшим решением будет писать код таким образом, чтобы гарантировать, что если будет вызван такой «просроченный» делегат, ничего плохого не случится.

Следующий пример демонстрирует, как создавать экземпляры делегатов, объединять, удалять и вызывать их:

```
using System;
delegate void D(int x);

class C
{
   public static void M1(int i)
   {
      Console.WriteLine("C.M1: " + i);
   }

   public static void M2(int i)
   {
      Console.WriteLine("C.M2: " + i);
   }

   public void M3(int i)
   {
      Console.WriteLine("C.M3: " + i);
   }
}

class Test
{
   static void Main()
   {
      D cd1 = new D(C.M1);
      cd1(-1);                    // Вызвать M1

      D cd2 = new D(C.M2);
      cd2(-2);                    // Вызвать M2

      D cd3 = cd1 + cd2;
      cd3(10);                    // Вызвать сначала M1, затем M2
            cd3 += cd1;
      cd3(20);                    // Вызвать M1, M2, затем M1
```

продолжение ☞

```
        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);                    // Вызвать M1, M2, M1, затем M3

        cd3 -= cd1;                 // Удалить последний M1
        cd3(40);                    // Вызвать M1, M2, затем M3

        cd3 -= cd4;
        cd3(50);                    // Вызвать M1, затем M2

        cd3 -= cd2;
        cd3(60);                    // Вызвать M1

        cd3 -= cd2;                 // Невозможность удаления не приводит к ошибке
        cd3(60);                    // Вызвать M1

        cd3 -= cd1;                 // Список вызова пуст, поэтому
                                    // cd3 имеет значение null

        //      cd3(70);            // Выбрасывается исключение
                                    // System.NullReferenceException

        cd3 -= cd1;                 // Невозможность удаления не приводит к ошибке
    }
}
```

Как показано в операторе `cd3 += cd1;`, делегат может входить в список вызова несколько раз. В этом случае он просто вызывается по одному разу для каждого вхождения. При удалении этого делегата из такого списка вызова фактически удаляется его последнее вхождение.

Непосредственно перед выполнением последнего оператора, `cd3 -= cd1;`, делегат `cd3` ссылается на пустой список вызова. Попытка удаления делегата из пустого списка (или удаления несуществующего делегата из непустого списка) не является ошибкой.

Результатом выполнения программы будет следующий вывод:

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

Глава 16

Исключения

Исключения в языке C# предоставляют структурированный, унифицированный и безопасный с точки зрения типов способ обработки ошибок как на системном уровне, так и на уровне приложения. Механизм исключений в C# достаточно сильно похож на используемый в C++, однако существуют несколько важных отличий.

> **ЭРИК ЛИППЕРТ**
>
> Касательно исключений системного уровня и уровня приложения сформировалось правило, заключавшееся в том, что все исключения уровня приложения должны были наследоваться от `ApplicationException`. Это оказалось плохой идеей, и мы больше не рекомендуем использовать данный подход. Теперь почти во всех случаях не имеет значения, пришло исключение из приложения или из библиотеки классов платформы.

- В C# все исключения должны быть представлены экземпляром класса, унаследованного от `System.Exception`. В C++ для представления исключения можно использовать значение любого типа.

> **ЭРИК ЛИППЕРТ**
>
> В первой версии CLR было возможным выбросить не-исключение и перехватить его в C#, используя блок `catch` без параметров. Это приводило к появлению запутанного кода обработки ошибок в программах на C#, которым требовалось перехватывать «ненастоящие» исключения. В новых версиях CLR выбрасываемый объект, не являющийся исключением, автоматически «обертывается» в объект-исключение, который может быть перехвачен обычным способом.

- В C# можно использовать блок `finally` (раздел 8.10) для написания завершающего кода, который выполняется как при нормальной работе программы, так и при возникновении исключительных ситуаций. В C++ это сложно сделать без дублирования кода.

> **ЭРИК ЛИППЕРТ**
>
> Иногда можно услышать миф о том, что «блок `finally` выполняется всегда». Разумеется, это не так. Блок `finally` не выполняется, если (1) во фрагменте кода, защищенном `try`, возникает бесконечный цикл, (2) программа завершается до начала выполнения блока `finally`, в ситуации «быстрого краха» («fail-fast») или прекращения выполнения программы администратором, (3) кто-то выдергивает кабель питания из розетки. Более
>
> *продолжение* ⇨

> того, существуют ситуации, когда вы не хотите, чтобы выполнялись блоки `finally`. Если исключение указывает на безнадежно испорченное состояние программы, при котором выполнение этих блоков только ухудшит ситуацию, иногда единственно верным вариантом будет быстрый крах и завершение процесса до того, как он успеет навредить еще больше.

- В C# исключения системного уровня, такие как переполнение, деление на ноль и разыменование `null`, имеют четко определенные классы исключений и находятся наравне с состояниями ошибок уровня приложения.

> **БИЛЛ ВАГНЕР**
>
> Этот пункт подчеркивает важное преимущество, а именно тот факт, что исключения в C# имеют много общего с исключениями в C++. Эта общность позволяет сообществу C# использовать все наработки, сделанные сообществом C++ (в особенности Дэйвом Абрахамом и Гербом Саттером), чтобы определить набор практических правил, делающих код более надежным перед лицом исключений.

> **КРИС СЕЛЛЗ**
>
> Наличие единого унифицированного способа взаимодействия и обработки ошибок является одним из значительных преимуществ платформы .NET, но иногда о нем забывают. Когда я вижу использующую .NET программу, в которой обмен информацией об ошибках реализован как-то по-другому, я считаю этот подход неправильным, пока мне не продемонстрируют обратное. Мне в голову приходит только один такой случай, но он относится к одной из спроектированных мной программ, поэтому меня сложно считать непредвзятым судьей.

> **ПИТЕР СЕСТОФТ**
>
> Одним большим исключением (простите за каламбур) из правила Криса является числовой код, где для сообщения об ошибках и их передачи используются `NaN`, а для кодирования информации об ошибке — биты полезной нагрузки в `NaN` (см. примечания в разделе 7.8.1). Выбрасывание исключения может быть на 3 или 4 порядка медленнее, чем передача `NaN`. Разумеется, скорость не имеет значения, если программа должна завершиться с сообщением об ошибке даже после одного исключения. Однако в некоторых ситуациях числовая «ошибка» может означать просто отсутствие входных данных, в результате чего в процессе вычислений могут происходить миллионы и миллиарды «ошибок»; в этом случае скорость имеет значение.

16.1. Причины исключений

Исключения могут выбрасываться по двум различным причинам.

- Оператор `throw` (раздел 8.9.5) немедленно и безусловно выбрасывает исключение. Управление никогда не передается оператору, следующему сразу за `throw`.
- Некоторые исключительные ситуации, возникающие в процессе обработки операторов и выражений C#, при определенных обстоятельствах могут сге-

нерировать исключение, если операция не может быть корректно завершена. Например, операция целочисленного деления (раздел 7.8.2) выбрасывает исключение `System.DivideByZeroException`, если делитель равен нулю. Список различных исключений, подобных этому, приведен в разделе 16.4.

> **КРИСТИАН НЕЙГЕЛ**
>
> Вы не должны выбрасывать исключения типа `Exception`; предпочтительнее использовать исключение унаследованного от `Exception` типа.

16.2. Класс System.Exception

Класс `System.Exception` представляет собой базовый тип для всех исключений. Он имеет ряд важных свойств, которые присутствуют во всех исключениях:

- `Message` — свойство типа `string`, доступное только для чтения, которое содержит описание причины исключения в удобной для человека форме.
- `InnerException` — свойство типа `Exception`, доступное только для чтения. Если его значение не равно `null`, оно ссылается на исключение, ставшее причиной текущего исключения, — это значит, что текущее исключение было сгенерировано в блоке `catch`, обрабатывавшем `InnerException`. Значение `null` у этого свойства означает, что данное исключение не было вызвано другим исключением. Количество объектов исключений в такой цепочке может быть произвольным.

Значения этих свойств могут быть указаны при вызове конструктора экземпляра `System.Exception`.

> **ЭРИК ЛИППЕРТ**
>
> Из приведенного описания можно предположить, что исключения являются неизменяемыми объектами, хотя на самом деле это не так. Каждый раз, когда вы выбрасываете конкретный экземпляр объекта исключения, состояние стека вызовов, сохраненное в этом исключении, сбрасывается. Например, если вы создаете один экземпляр исключения, а потом выбрасываете его в нескольких потоках, изменения в стеке вызовов исключения будут видны во всех обрабатывающих его потоках, что, скорее всего, будет сбивать вас с толку. Создавайте новое исключение каждый раз, когда оно вам требуется.

16.3. Как обрабатываются исключения

Исключения обрабатываются оператором `try` (раздел 8.10).

Когда возникает исключение, система ищет ближайший оператор `catch`, способный его обработать, в соответствии с типом времени выполнения исключения. В первую очередь в текущем методе осуществляется поиск лексически охватывающего оператора `try`, после чего по порядку рассматриваются соответствующие ему операторы `catch`. Если найти подходящий не удается, в методе, вызвавшем

текущий метод, выполняется поиск лексически охватывающего оператора `try`, который содержит точку вызова текущего метода. Поиск продолжается до тех пор, пока не будет найден оператор `catch`, способный обработать текущее исключение, то есть в котором в качестве класса исключения указан такой же класс, к которому относится тип времени выполнения выбрасываемого исключения, или его базовый класс. Оператор `catch`, в котором не указан класс исключения, может обрабатывать любое исключение.

Когда найден подходящий оператор `catch`, система подготавливает передачу управления первому оператору в блоке `catch`. Перед началом выполнения блока `catch` система сначала последовательно выполняет все блоки `finally`, соответствующие операторам `try`, вложенным в тот, который перехватил исключение.

Если подходящий оператор `catch` не найден, происходит одно из двух событий:

- Если поиск подходящего оператора `catch` достигает статического конструктора (раздел 10.12) или инициализатора статического поля, в точке вызова статического конструктора выбрасывается исключение `System.TypeInitializationException`. Исходное исключение указывается в качестве внутреннего исключения для `System.TypeInitializationException`.
- Если поиск подходящих операторов `catch` достигает кода, который изначально запустил поток, выполнение потока завершается. Последствия такого завершения определяются реализацией.

ЭРИК ЛИППЕРТ

В ранних версиях CLR определенным реализацией поведением при возникновении в главном потоке необработанного исключения было завершение выполнения программы. Если исключение возникало в дополнительном потоке, этот поток завершался, однако главный поток продолжал выполняться. Такая стратегия оказалась неудачной, поскольку приложения, содержащие ошибки в коде дополнительных потоков, постепенно теряли все эти потоки, и в итоге оставалось работающее приложение, которое ничего не делало, что приводило пользователей в замешательство. Новая стратегия более агрессивна: необработанное исключение в *любом* потоке приводит к завершению приложения. Зачастую лучше аварийно завершить работу программы, чем продолжать выполнение, как будто ничего не произошло; после достаточного количества таких завершений кто-нибудь в конечном счете найдет и решит проблему.

Отдельного упоминания заслуживают исключения, возникающие при выполнении деструктора. Если во время выполнения деструктора происходит исключение и оно не перехватывается, выполнение деструктора завершается и вызывается деструктор базового класса (если он есть). Если базового класса нет (как в случае с типом `object`) или в нем отсутствует деструктор, исключение игнорируется.

КРИСТИАН НЕЙГЕЛ

Компилятор C# создает из деструктора финализатор. Внутрь финализатора добавляется блок `try-finally`, и финализатор базового класса вызывается в блоке `finally`.

16.4. Распространенные классы исключений

Следующие исключения выбрасываются некоторыми операциями C#.

Исключение	Описание
`System.ArithmeticException`	Базовый класс для исключений, возникающих при арифметических операциях, таких как System.DivideByZeroException и System.OverflowException
`System.ArrayTypeMismatchException`	Выбрасывается при невозможности добавления элемента в массив из-за того, что фактический тип элемента несовместим с фактическим типом массива
`System.DivideByZeroException`	Выбрасывается при попытке разделить целочисленное значение на ноль
`System.IndexOutOfRangeException`	Выбрасывается при попытке обратиться к элементу массива по индексу, который является отрицательным или выходит за границы массива
`System.InvalidCastException`	Выбрасывается, когда явное преобразование базового типа или интерфейса в унаследованный тип во время выполнения программы завершается неудачей
`System.NullReferenceException`	Выбрасывается, когда ссылка, имеющая значение `null`, используется таким образом, при котором требуется обращение к объекту, на который она должна ссылаться
`System.OutOfMemoryException`	Выбрасывается при невозможности выделить память (операцией `new`)
`System.OverflowException`	Выбрасывается, когда арифметическая операция, выполняемая в проверяемом контексте (`checked`), приводит к переполнению
`System.StackOverflowException`	Выбрасывается, когда в стеке выполнения не остается места вследствие большого числа незавершенных вызовов методов; обычно является признаком слишком глубокой или неограниченной рекурсии
`System.TypeInitializationException`	Выбрасывается, когда статический конструктор выбрасывает исключение и отсутствуют операторы `catch`, способные его перехватить

ДЖОН СКИТ

В этой таблице перечислены не наиболее часто встречающиеся исключения, а скорее исключения, которые могут естественным образом возникать в среде выполнения как результат выполнения операций C#. Вы должны явно выбрасывать эти исключения как можно реже (или вообще никогда). И наоборот, наиболее часто встречающиеся исключения, такие как `System.ArgumentException` и его подклассы, здесь не перечислены — они выбрасываются в коде более высокого уровня, будь то системные библиотеки или приложения.

ЭРИК ЛИППЕРТ

Я разделяю исключения на четыре категории: *фатальные*, *глупые*, *досаждающие* и *внесистемные*.

Фатальными называются те исключения, которые вы никогда не выбрасываете и не перехватываете; сюда входят нехватка памяти, аварийное завершение потока и т. д. Вы не создавали эту проблему и не можете ее решить; исключение — всего лишь механизм, оповещающий вас о приближающемся конце света.

Глупых исключений можно избежать, и поэтому они выбрасываются только некорректно написанными программами; сюда входят разыменование нулевой ссылки, неправильный аргумент и т. д. Никогда не перехватывайте такие исключения; перехват глупого исключения означает сокрытие чьей-то ошибки. Вместо этого исправьте ее.

Досаждающими называются исключения, которые вам приходится обрабатывать, поскольку API спроектирован таким образом, что использует исключения для взаимодействия. Например, в предыдущих версиях платформы отсутствовал метод «Является ли строка корректным GUID?»; для ответа на этот вопрос вы могли вызвать конструктор GUID и перехватывать досаждающее исключение, чтобы определить, успешно ли он выполнился или нет. Старайтесь не писать код, который выбрасывает досаждающие исключения.

Внесистемными называются такие исключения, которые вы обязаны перехватывать, поскольку они сообщают вам неожиданные факты о реальном мире: извлечение CD из привода, отключение сетевого маршрутизатора и т. д.

Глава 17
Атрибуты

Во многих случаях язык C# позволяет программисту указывать описательную информацию для определенных в программе сущностей. Например, доступность метода класса указывается путем добавления к нему *модификаторов-метода* `public`, `protected`, `internal` и `private`.

C# дает программистам возможность создавать новые типы описательной информации, называемые **атрибутами**. Впоследствии программисты могут прикреплять атрибуты к различным сущностям программы и получать из атрибутов информацию в среде выполнения. Например, платформа может определять атрибут `HelpAttribute`, который можно добавлять к некоторым элементам программы (таким как классы и методы), чтобы установить соответствие между этими элементами и их документацией.

ДЖЕСС ЛИБЕРТИ

Распространенное и эффективное использование атрибутов можно увидеть в Silverlight и WPF, где они используются для обозначения состояний вида (view states), ассоциированных с классом.
Схожим образом атрибуты используются в библиотеках тестирования для отделения тестирующих методов от вспомогательных. Веб-сервисы используют атрибуты для обозначения методов, доступных клиентам.

Атрибуты определяются посредством объявления классов атрибутов (раздел 17.1), которые могут иметь позиционные и именованные параметры (раздел 17.12). Атрибуты прикрепляются к объектам программы на C# с помощью спецификаций атрибута (раздел 17.2) и во время выполнения программы могут быть получены в виде экземпляров атрибута (раздел 17.3).

ЭРИК ЛИППЕРТ

Старайтесь использовать атрибуты только чтобы говорить о *самом типе*, а не для обозначения *семантики* этого типа. Предположим, к примеру, что у вас есть класс `Book`, содержащий свойство `Author`. Это часть семантики класса: класс представляет книгу, а у книг есть авторы. Если вы добавите к классу `Book` атрибут `AuthorAttribute`, он будет обозначать не автора *книги*, а скорее автора *класса*. Класс `Television` может содержать свойство `Obsolete`, означающее, что определенная модель снята с производства. Если вы добавите атрибут `ObsoleteAttribute` к классу, это будет означать, что устаревшим является *сам класс*, а не *конкретное устройство*, представленное этим классом.

17.1. Классы атрибутов

Класс, прямо или косвенно унаследованный от абстрактного класса `System.Attribute`, является **классом атрибута**. Объявление класса атрибута определяет новый тип **атрибута**, который можно поместить в объявление. По традиции к названиям классов атрибутов добавляется суффикс `Attribute`. При использовании атрибутов этот суффикс можно опускать.

17.1.1. Использование атрибутов

Для описания того, как может использоваться класс атрибута, предназначен атрибут `AttributeUsage` (раздел 17.4.1).

`AttributeUsage` принимает позиционный параметр (раздел 17.1.2), позволяющий указать типы объявлений, в которых можно использовать класс атрибута. Пример:

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

Здесь определен класс атрибута с именем `SimpleAttribute`, который можно добавлять только к *объявлениям-классов* и *объявлениям-интерфейсов*. В примере

```
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

продемонстрированы некоторые способы использования атрибута `Simple`. Хотя он был определен с именем `SimpleAttribute`, при использовании суффикс `Attribute` можно опускать, в результате чего имя сокращается до `Simple`. Таким образом, приведенный выше пример семантически эквивалентен следующему:

```
[SimpleAttribute] class Class1 {...}
[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` принимает именованный параметр (раздел 17.1.2) `AllowMultiple`, определяющий, можно ли указывать атрибут более одного раза для конкретной сущности. Если `AllowMultiple` для класса атрибута имеет значение `true`, этот класс атрибута называется *классом атрибута многократного использования* и может быть указан для сущности более одного раза. Если `AllowMultiple` имеет значение `false` или не указан, класс атрибута называется *классом атрибута однократного использования* и его можно указывать для сущности не более одного раза. Пример:

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
   private string name;
```

```
   public AuthorAttribute(string name) {
       this.name = name;
   }

   public string Name {
       get { return name; }
   }
}
```

Здесь определен класс атрибута многократного использования с именем `AuthorAttribute`. В примере

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

продемонстрировано объявление класса, использующее атрибут `Author` дважды.

> **КРИС СЕЛЛЗ**
>
> Для тех, кто читает этот пример и не знает, кто такие Брайан Керниган и Деннис Ритчи: позор! Они — настоящие легенды в области программного обеспечения.

> **ДЖОН СКИТ**
>
> Хотя я и согласен с Крисом, я, безусловно, надеюсь, что ни один из этих достопочтенных джентльменов никогда не написал бы класс с именем `Class1`.

У `AttributeUsage` есть еще один именованный параметр с именем `Inherited`, который определяет, будет ли атрибут, указанный для базового класса, наследоваться производными от него классами. Если параметр `Inherited` для класса-атрибута имеет значение `true`, этот атрибут наследуется. Если он имеет значение `false`, этот атрибут не наследуется. Если параметр не указан, значением по умолчанию является `true`.

Класс атрибута `X`, к которому не прикреплен атрибут `AttributeUsage`, как в примере

```
using System;

class X: Attribute {...}
```

эквивалентен следующему классу:

```
using System;

[AttributeUsage(
   AttributeTargets.All,
   AllowMultiple = false,
   Inherited = true)
]
class X: Attribute {...}
```

17.1.2. Позиционные и именованные параметры

> **БИЛЛ ВАГНЕР**
>
> В большинстве случаев использование именованных параметров делает программу более ясной и понятной. Позиционные параметры следует использовать только для атрибутов с одним свойством, смысл которого понятен из имени атрибута, например `AuthorAttribute`.

Классы атрибутов могут иметь **позиционные параметры** и **именованные параметры**. Каждый открытый конструктор экземпляра в классе атрибута определяет допустимую последовательность позиционных параметров для этого класса атрибута. Каждое нестатическое открытое доступное для чтения и записи поле и свойство в классе атрибута определяет именованный параметр для класса атрибута.

Пример:

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
   public HelpAttribute(string url) {  // Позиционный параметр
       ...
   }

   public string Topic {                 // Именованный параметр
      get {...}
      set {...}
   }

   public string Url {
       get {...}
   }
}
```

Здесь определен класс атрибута с именем `HelpAttribute`, который имеет один позиционный параметр, `url`, и один именованный параметр, `Topic`. Несмотря на то что свойство `Url` является нестатическим и открытым, оно не определяет именованный параметр, так как доступно только для чтения, но не для записи.

Этот класс атрибута может быть использован следующим образом:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

17.1.3. Типы параметров атрибута

Типы позиционных и именованных параметров для класса атрибута ограничены следующими **типами параметров атрибута**:

- Один из следующих типов: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.
- Тип `object`.
- Тип `System.Type`.
- Перечислимый тип, при условии, что он сам, а также все типы, в которые он вложен (если такие имеются), имеют открытый вид доступа (раздел 17.2).
- Одномерные массивы вышеуказанных типов.

ДЖОН СКИТ

Заметьте, что `decimal` не может быть использован в качестве типа параметра, хотя вы можете объявлять константы типа `decimal`. Это один из примеров проникновения правил CLI в C#.

МАРЕК САФАР

Несмотря на то что использование массивов поддерживается, для атрибутов ссылочных типов не допускается ковариантность массивов. Аналогично, существующий в C# 4.0 тип `dynamic` не может использоваться там, где ожидается тип `object`.

Аргумент конструктора или открытое поле, имеющие типы, отличные от перечисленных, не могут использоваться в качестве позиционного или именованного параметра в спецификации атрибута.

17.2. Спецификация атрибута

Спецификация атрибута — это применение определенного ранее атрибута к объявлению. Атрибут является частью дополнительной декларативной информации, которая указывается для объявления. Атрибуты могут быть заданы в глобальной области видимости (чтобы указать атрибут для содержащей его сборки или модуля), а также для *объявлений-типов* (раздел 9.6), *объявлений-элементов-класса* (раздел 10.1.5), *объявлений-элементов-интерфейса* (раздел 13.2), *объявлений-элементов-структуры* (раздел 11.2), *объявлений-элементов-перечислений* (раздел 14.3), *объявлений-кодов-доступа* (раздел 10.7.2), *объявлений-кодов-доступа-для-событий* (раздел 10.8.1) и *списков-формальных-параметров* (раздел 10.6.1).

Атрибуты указываются в **секциях атрибутов**. Секция атрибутов состоит из пары квадратных скобок, в которые заключен список из одного или нескольких

атрибутов, разделенных запятыми. Порядок перечисления атрибутов в этом списке, а также порядок, в котором секции атрибутов прикрепляются к одной программной сущности, не имеет значения. Например, спецификации атрибутов `[A][B]`, `[B][A]`, `[A, B]` и `[B, A]` эквивалентны.

глобальные-атрибуты:
  секции-глобальных-атрибутов

секции-глобальных-атрибутов:
  секция-глобальных-атрибутов
  секции-глобальных-атрибутов секция-глобальных-атрибутов

секция-глобальных-атрибутов:
  [*спецификатор-целевого-объекта-глобального-атрибута список-атрибутов* **]**
  [*спецификатор-целевого-объекта-глобального-атрибута список-атрибутов* **,]**

спецификатор-целевого-объекта-глобального-атрибута
  целевой-объект-глобального-атрибута **:**

целевой-объект-глобального-атрибута:
  assembly
  module

атрибуты:
  секции-атрибутов

секции-атрибутов:
  секция-атрибутов
  секции-атрибутов секция-атрибутов

секция-атрибутов:
  [*спецификатор-целевого-объекта-атрибута*$_{opt}$ *список-атрибутов* **]**
  [*спецификатор-целевого-объекта-атрибута*$_{opt}$ *список-атрибутов* **,]**

спецификатор-целевого-объекта-атрибута:
  целевой-объект-атрибута **:**

целевой-объект-атрибута:
  field
  event
  method
  param
  property
  return
  type

список-атрибутов:
  атрибут
  список-атрибутов **,** *атрибут*

атрибут:
  имя-атрибута аргументы-атрибута$_{opt}$

имя-атрибута:
  имя-типа

```
аргументы-атрибута:
   (   список-позиционных-аргументов_opt   )
   (   список-позиционных-аргументов   ,   список-именованных-аргументов   )
   (   список-именованных-аргументов   )

список-позиционных-аргументов:
   позиционный-аргумент
   список-позиционных-аргументов   ,   позиционный-аргумент

позиционный-аргумент:
   имя-аргумента_opt   выражение-аргумента-атрибута

список-именованных-аргументов:
   именованный-аргумент
   список-именованных-аргументов   ,   именованный-аргумент

именованный-аргумент:
   идентификатор   =   выражение-аргумента-атрибута

выражение-аргумента-атрибута:
   выражение
```

Атрибут состоит из *имени-атрибута* и необязательного списка позиционных и именованных аргументов. Позиционные аргументы (если они есть) находятся перед именованными. Позиционный аргумент состоит из *выражения-аргумента-атрибута*; именованный аргумент состоит из имени, за которым следуют знак равенства, а затем *выражение-аргумента-атрибута*, которые все вместе подчиняются тем же правилам, что и простое присваивание. Порядок именованных аргументов не имеет значения.

Имя-атрибута определяет класс атрибута. Если *имя-атрибута* имеет форму *имя-типа*, это имя должно ссылаться на класс атрибута. В противном случае возникает ошибка компиляции. Пример:

```
class Class1 {}
[Class1] class Class2 {}    // Ошибка
```

Этот код приводит к ошибке компиляции, поскольку предпринята попытка использовать в качестве класса атрибута `Class1`, который не является классом атрибута.

В некоторых контекстах допускается спецификация атрибута для более чем одного целевого объекта. Программа может явно задать целевой объект с помощью *спецификатора-целевого объекта*. Когда атрибут размещается на глобальном уровне, требуется *спецификатор-целевого-объекта-глобального-атрибута*. При размещении в остальных позициях используются приемлемые умолчания, но с помощью *спецификатора-целевого-объекта-атрибута* их можно подтвердить или переопределить для некоторых двусмысленных случаев (или просто подтвердить для недвусмысленных случаев). Таким образом, *спецификаторы-целевого-объекта-атрибута* обычно можно опускать, за исключением атрибутов глобального уровня. Потенциально двусмысленные ситуации разрешаются следующим образом:

- Атрибут, указанный в глобальной области видимости, может применяться или к целевой сборке, или к целевому модулю. В этом контексте отсутствует

значение по умолчанию, поэтому здесь всегда требуется *спецификатор-целевого-объекта-атрибута*. Наличие спецификатора `assembly` означает, что атрибут применяется к целевой сборке, а наличие спецификатора `module` означает, что атрибут применяется к целевому модулю.

- Атрибут, указанный в объявлении делегата, может применяться или к объявляемому делегату, или к возвращаемому им значению. При отсутствии *спецификатора-целевого-объекта-атрибута* атрибут применяется к делегату. Наличие спецификатора `type` означает, что атрибут применяется к делегату, а наличие спецификатора `return` означает, что атрибут применяется к возвращаемому значению.
- Атрибут, указанный в объявлении метода, может применяться или к объявляемому методу, или к возвращаемому им значению. При отсутствии *спецификатора-целевого-объекта-атрибута* атрибут применяется к методу. Наличие спецификатора `method` означает, что атрибут применяется к методу, а наличие спецификатора `return` означает, что атрибут применяется к возвращаемому значению.
- Атрибут, указанный в объявлении операции, может применяться или к объявляемой операции, или к возвращаемому ей значению. При отсутствии *спецификатора-целевого-объекта-атрибута* атрибут применяется к операции. Наличие спецификатора `method` означает, что атрибут применяется к операции, а наличие спецификатора `return` означает, что атрибут применяется к возвращаемому значению.
- Атрибут, указанный в объявлении события, не включающем в себя коды доступа события, может применяться к объявляемому событию, к ассоциированному с ним полю (если событие не является абстрактным) или к ассоциированным с ним методам добавления (`add`) и удаления (`remove`). При отсутствии *спецификатора-целевого-объекта-атрибута* атрибут применяется к событию. Наличие спецификатора `event` означает, что атрибут применяется к событию; наличие спецификатора `field` означает, что атрибут применяется к полю, а наличие спецификатора `method` означает, что атрибут применяется к методам.
- Атрибут, указанный для кода доступа `get` в объявлении свойства или индексатора, может применяться или к ассоциированному методу, или к возвращаемому им значению. При отсутствии *спецификатора-целевого-объекта-атрибута* атрибут применяется к методу. Наличие спецификатора `method` означает, что атрибут применяется к методу, а наличие спецификатора `return` означает, что атрибут применяется к возвращаемому значению.
- Атрибут, указанный для кода доступа `set` в объявлении свойства или индексатора, может применяться или к ассоциированному методу, или к его единственному неявному параметру. При отсутствии *спецификатора-целевого-объекта-атрибута* атрибут применяется к методу. Наличие спецификатора `method` означает, что атрибут применяется к методу; наличие спецификатора `param` означает, что атрибут применяется к параметру, а наличие спецификатора `return` означает, что атрибут применяется к возвращаемому значению.

- Атрибут, указанный для кода доступа добавления или удаления в объявлении события, может применяться или к ассоциированному методу, или к его единственному параметру. При отсутствии *спецификатора-целевого-объекта-атрибута* атрибут применяется к методу. Наличие спецификатора `method` означает, что атрибут применяется к методу; наличие спецификатора `param` означает, что атрибут применяется к параметру, а наличие спецификатора `return` означает, что атрибут применяется к возвращаемому значению.

В других контекстах добавление *спецификатора-целевого-объекта-атрибута* допустимо, но не является необходимым. Например, объявление класса может или включать, или не включать спецификатор `type`:

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

Указание некорректного *спецификатора-целевого-объекта-атрибута* приводит к ошибке. Например, спецификатор `param` не может быть использован в объявлении класса:

```
[param: Author("Brian Kernighan")]       // Ошибка
class Class1 {}
```

По традиции к именам классов атрибутов добавляется суффикс `Attribute`. В *имени-атрибута*, имеющем вид *имя-типа*, этот суффикс может присутствовать или быть опущен. Если класс атрибута обнаруживается как при наличии суффикса, так и при его отсутствии, возникает двусмысленность, приводящая к ошибке компиляции. Если *имя-атрибута* записывается так, что его крайний правый *идентификатор* является *буквальным идентификатором* (раздел 2.4.2), то используется только атрибут без суффикса, и, таким образом, двусмысленность разрешается. Пример:

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                                // Ошибка: двусмысленность
class Class1 {}

[XAttribute]                       // Ссылается на XAttribute
class Class2 {}

[@X]                               // Ссылается на X
class Class3 {}

[@XAttribute]                      // Ссылается на XAttribute
class Class4 {}
```

Здесь показаны два класса атрибутов с именами X и XAttribute. Атрибут [X] является двусмысленным, поскольку он может ссылаться как на X, так и на XAttribute. В таких редких случаях использование буквального идентификатора позволяет указать точный смысл атрибута. Атрибут [XAttribute] не является двусмысленным (хотя мог бы, если бы существовал класс атрибута с именем XAttributeAttribute!). Если удалить объявление класса X, оба атрибута будут ссылаться на класс с именем XAttribute, как показано в примере:

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                             // Ссылается на XAttribute
class Class1 {}

[XAttribute]                    // Ссылается на XAttribute
class Class2 {}

[@X]                            // Ошибка: атрибут с именем "X" отсутствует
class Class3 {}
```

При указании класса атрибута однократного использования более одного раза для одной сущности возникает ошибка компиляции. Пример:

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
   string value;

   public HelpStringAttribute(string value) {
       this.value = value;
   }

   public string Value {
       get {...}
   }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]

public class Class1 {}
```

Этот код приводит к ошибке компиляции, так как в объявлении Class1 предпринята попытка более одного раза использовать HelpString, являющийся классом атрибута однократного использования.

Выражение E является *выражением-аргумента-атрибута*, если выполняются все перечисленные условия:

- Тип E является типом параметра атрибута (раздел 17.1.3).

- Во время компиляции значение `E` может быть разрешено в значение одного из следующих типов:
    - Константное значение.
    - Объект `System.Type`.
    - Одномерный массив *выражений-аргумента-атрибута*.

  Например:

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
   public int P1 {
      get {...}
      set {...}
   }

   public Type P2 {
      get {...}
      set {...}
   }

   public object P3 {
      get {...}
      set {...}
   }
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

Выражение-typeof (раздел 7.6.11), использованное в качестве выражения аргумента атрибута, может ссылаться на не являющийся обобщенным тип, закрытый (closed) сконструированный тип или неограниченный обобщенный тип, но не может ссылаться на открытый (open) тип. Это правило гарантирует, что выражение может быть разрешено на этапе компиляции.

```
class A: Attribute
{
    public A(Type t) {...}
}

class G<T>
{
    [A(typeof(T))] T t;                // Ошибка: открытый тип в атрибуте
}

class X
{
   [A(typeof(List<int>))] int x;   // Все в порядке: закрытый
                                   // сконструированный тип
   [A(typeof(List<>))] int y;      // Все в порядке: неограниченный
                                   // обобщенный тип
}
```

17.3. Экземпляры атрибутов

Экземпляр атрибута — это экземпляр, представляющий атрибут во время выполнения программы. Атрибут определяется классом атрибута, позиционными аргументами и именованными аргументами. Экземпляр атрибута является экземпляром класса атрибута, инициализируемым позиционными и именованными аргументами.

Процесс получения экземпляра атрибута включает обработку и на этапе компиляции, и во время выполнения, как описано в следующих разделах.

17.3.1. Компиляция атрибута

Компиляция *атрибута* с классом-атрибутом `T`, *списком-позиционных-аргументов* `P` и *списком-именованных аргументов* `N` состоит из следующих шагов:

- Выполнить действия по обработке на этапе компиляции для компиляции *выражения-создания-объекта*, имеющего форму `new T(P)`. В результате выполнения этих действий или возникнет ошибка компиляции, или для `T` будет определен конструктор экземпляра `C`, который может быть вызван во время выполнения программы.
- Если `C` имеет вид доступа, отличный от открытого, возникает ошибка компиляции.
- Для каждого *именованного-аргумента* `Arg` в `N`:
  - Пусть `Name` будет *идентификатором именованного-аргумента* `Arg`.
  - `Name` должен определять нестатическое открытое поле или свойство `T`, доступное для чтения и записи. Если у `T` нет такого поля или свойства, возникает ошибка компиляции.
- Для создания экземпляра атрибута во время выполнения программы сохранить следующую информацию: класс атрибута `T`, конструктор экземпляра `C` для `T`, *список-позиционных-аргументов* `P` и *список-именованных-аргументов* `N`.

> **МАРЕК САФАР**
>
> Ограничение на открытый вид доступа в данном случае может немного сбивать с толку. Например, атрибуты класса с лексической точки зрения объявляются за пределами класса, но они все равно могут использовать закрытые константы, объявленные внутри этого класса.

17.3.2. Получение экземпляра атрибута во время выполнения

В результате компиляции *атрибута* получается класс атрибута `T`, конструктор экземпляра `C` для `T`, *список-позиционных-аргументов* `P` и *список-именованных-*

аргументов `N`. При наличии этой информации во время выполнения экземпляр атрибута может быть получен после следующих шагов:

- Выполнить действия по обработке во время выполнения для выполнения *выражения-создания-объекта*, имеющего форму `new T(P)`, используя конструктор экземпляра `C`, определенный на этапе компиляции. В результате этих действий или возникнет исключение, или будет получен экземпляр `O` класса `T`.
- Для каждого *именованного-аргумента* `Arg` в `N`, по порядку:
  - Пусть `Name` будет *идентификатором именованного-аргумента* `Arg`. Если `Name` не определяет нестатическое открытое поле или свойство `O`, доступное для чтения и записи, выбрасывается исключение.
  - Пусть `Value` будет результатом вычисления *выражения-аргумента-атрибута* для `Arg`.
  - Если `Name` определяет поле в `O`, установить для этого поля значение `Value`.
  - В противном случае `Name` определяет свойство в `O`. Установить значение `Value` для этого свойства.
  - Результатом является `O` — экземпляр класса-атрибута `T`, инициализированный *списком-позиционных-аргументов* `P` и *списком-именованных-аргументов* `N`.

17.4. Зарезервированные атрибуты

Небольшое количество атрибутов тем или иным образом оказывают влияние на язык. Сюда входят:

- `System.AttributeUsageAttribute` (раздел 17.4.1), который предназначен для описания способов использования класса атрибута.
- `System.Diagnostics.ConditionalAttribute` (раздел 17.4.2), который используется для определения условных методов.
- `System.ObsoleteAttribute` (раздел 17.4.3), используемый для того, чтобы помечать элемент как устаревший.

17.4.1. Атрибут AttributeUsage

Атрибут `AttributeUsage` предназначен для описания способов использования класса атрибута.

Класс, к которому применен атрибут `AttributeUsage`, должен прямо или косвенно наследоваться от класса `System.Attribute`. В противном случае возникает ошибка компиляции.

```
namespace System
{
   [AttributeUsage(AttributeTargets.Class)]
   public class AttributeUsageAttribute: Attribute
```

продолжение ☞

```
   {
      public AttributeUsageAttribute(AttributeTargets validOn) {
                  ...
              }

      public virtual bool AllowMultiple { get {...} set {...} }

      public virtual bool Inherited { get {...} set {...} }

      public virtual AttributeTargets ValidOn { get {...} }
   }
   public enum AttributeTargets
   {
      Assembly = 0x0001,
      Module = 0x0002,
      Class = 0x0004,
      Struct = 0x0008,
      Enum = 0x0010,
      Constructor = 0x0020,
      Method = 0x0040,
      Property = 0x0080,
      Field = 0x0100,
      Event = 0x0200,
      Interface = 0x0400,
      Parameter = 0x0800,
      Delegate = 0x1000,
      ReturnValue = 0x2000,

      All = Assembly | Module | Class | Struct | Enum |
              Constructor | Method | Property | Field | Event |
              Interface | Parameter | Delegate | ReturnValue
   }
}
```

17.4.2. Атрибут Conditional

Атрибут `Conditional` делает возможным определение **условных методов** и **классов условных атрибутов**.

```
namespace System.Diagnostics
{
   [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
           AllowMultiple = true)]
   public class ConditionalAttribute: Attribute
   {
      public ConditionalAttribute(string conditionString) {...}
      public string ConditionString { get {...} }
   }
}
```

17.4.2.1. Условные методы

Метод, к которому применен атрибут `Conditional`, является условным методом. Атрибут `Conditional` указывает на существование условия, проверяющего символ

условной компиляции. Вызовы условного метода или оставляются, или опускаются в зависимости от того, определен ли этот символ в точке вызова. Если символ определен, вызов оставляется; в противном случае вызов (включая вычисление получателя и параметров вызова) опускается.

На условный метод накладываются следующие ограничения:

- Условный метод должен находиться в *объявлении-класса* или *объявлении-структуры*. Если атрибут `Conditional` применяется к методу, определенному в объявлении интерфейса, возникает ошибка компиляции.
- Типом возвращаемого условным методом значения должен являться `void`.
- У условного метода должен отсутствовать модификатор `override`. В то же время использование модификатора `virtual` допустимо. Методы, переопределяющие такой метод, являются условными неявно и не должны быть явно отмечены атрибутом `Conditional`.
- Условный метод не может являться реализацией метода интерфейса. В противном случае возникает ошибка компиляции.

Помимо этого ошибка компиляции также возникает, если условный метод используется в *выражении-создания-делегата*. Пример:

```
#define DEBUG

using System;
using System.Diagnostics;
class Class1
{
   [Conditional("DEBUG")]
   public static void M() {
       Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
   public static void Test() {
       Class1.M();
   }
}
```

Здесь метод `Class1.M` объявлен как условный. Этот метод вызывается в методе `Test` класса `Class2`. Так как символ условной компиляции `DEBUG` определен, при вызове метода `Class2.Test` будет вызван метод `M`. Если бы символ `DEBUG` определен не был, метод `Class2.Test` не вызвал бы метод `Class1.M`.

Важно отметить, что включение или исключение вызова условного метода определяется символами условной компиляции в точке вызова. Рассмотрим пример.

Файл `class1.cs`:

```
using System.Diagnostics;

class Class1
{
   [Conditional("DEBUG")]
   public static void F() {
```

продолжение ⇨

```
    Console.WriteLine("Executed Class1.F");
    }
}
```

Файл `class2.cs`:

```
#define DEBUG

class Class2
{
   public static void G() {
           Class1.F();                    // F вызывается
   }
}
```

Файл `class3.cs`:

```
#undef DEBUG

class Class3
{
   public static void H() {
           Class1.F();                    // F не вызывается
   }
}
```

В этом примере классы `Class2` и `Class3` содержат вызовы условного метода `Class1.F`, условием для вызова которого является наличие символа `DEBUG`. Так как этот символ определен в контексте класса `Class2`, но не определен в контексте класса `Class3`, вызов `F` в `Class2` оставляется, а в `Class3` — опускается.

Использование условных методов в цепочке наследования может сбивать с толку. Вызовы условного метода, сделанные через `base` в форме `base.M`, подчиняются обычным правилам вызова условных методов.

Рассмотрим пример.

Файл `class1.cs`:

```
using System;
using System.Diagnostics;

class Class1
{
   [Conditional("DEBUG")]
   public virtual void M() {
           Console.WriteLine("Выполняется Class1.M");
   }
}
```

Файл `class2.cs`:

```
using System;

class Class2: Class1
{
   public override void M() {
   Console.WriteLine("Выполняется Class2.M");
           base.M();           // base.M не вызывается!
   }
}
```

Файл `class3.cs`:

```
#define DEBUG

using System;

class Class3
{
   public static void Test() {
      Class2 c = new Class2();
      c.M();                // M вызывается
   }
}
```

Здесь `Class2` содержит вызов метода `M`, определенного в его базовом классе. Этот вызов опускается, так как метод базового класса является условным и зависит от наличия символа `DEBUG`, который не определен. Таким образом, метод выведет на консоль только текст «`Выполняется Class2.M`». Благоразумное использование *директив-объявлений (pp-declarations)* может устранить подобные проблемы.

17.4.2.2. Классы условных атрибутов

Класс атрибута (раздел 17.1), к которому применен один или более атрибут `Conditional`, является *классом условного атрибута*. Таким образом, класс условного атрибута ассоциирован с символами условной компиляции, объявленными в его атрибутах `Conditional`. Пример:

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

Здесь `TestAttribute` объявлен как класс условного атрибута, ассоциированный с символами условной компиляции `ALPHA` и `BETA`.

Спецификации атрибута (раздел 17.2) для условного атрибута включаются, если один или более ассоциированных с ним символов условной компиляции определены в точке спецификации; в противном случае спецификация атрибута опускается.

Важно отметить, что включение или исключение спецификации атрибута для класса условного атрибута определяется символами условной компиляции в точке спецификации. Пример:

Файл `test.cs`:

```
using System;
using System.Diagnostics;

[Conditional("DEBUG")]
public class TestAttribute : Attribute {}
```

продолжение ⇗

Файл `class1.cs`:

```
#define DEBUG
[Test]
class Class1 {}                    // TestAttribute указывается
```

Файл `class2.cs`:

```
#undef DEBUG

[Test]
class Class2 {}                    // TestAttribute не указывается
```

Здесь к классам `Class1` и `Class2` применен атрибут `Test`, который является условным и зависит от того, определен ли символ `DEBUG`. Так как этот символ определен в контексте класса `Class1` и не определен в контексте класса `Class2`, спецификация атрибута `Test` для `Class1` оставляется, а для `Class2` — опускается.

ЭРИК ЛИППЕРТ

Условный атрибут, несомненно, имеет сильную связь с символами условной компиляции, но важно помнить, что они очень разные. Распространенной ошибкой является написание чего-то вроде этого:

```
#if DEBUG
int counter;
#endif
[Conditional("DEBUG")] void DoIt(int x) { ... }
...
DoIt(this.counter);
```

Определение поля `counter` полностью удаляется из программы, если символ `DEBUG` не определен. Откуда компилятор знает, что надо удалить вызов `DoIt`? Из того, что метод `DoInt(int)` является условным, а символ `DEBUG` не определен. Но откуда компилятор знает, что вызывается метод `DoIt(int)`, а не какой-нибудь перегруженный метод? Из того, что `counter` объявлен как `int` — ой, подождите, *определение этого поля было удалено*. Такого поля не существует, поэтому в неотладочной сборке компиляция завершится с ошибкой.

17.4.3. Атрибут Obsolete

Атрибут `Obsolete` используется для того, чтобы отмечать типы и элементы типов, которые не следует более использовать.

```
namespace System
{
   [AttributeUsage(
      AttributeTargets.Class |
      AttributeTargets.Struct |
      AttributeTargets.Enum |
      AttributeTargets.Interface |
      AttributeTargets.Delegate |
      AttributeTargets.Method |
      AttributeTargets.Constructor |
      AttributeTargets.Property |
      AttributeTargets.Field |
      AttributeTargets.Event,
```

```
      Inherited = false)
   ]
   public class ObsoleteAttribute : Attribute
   {
      public ObsoleteAttribute() {...}

      public ObsoleteAttribute(string message) {...}

      public ObsoleteAttribute(string message, bool error) {...}

      public string Message { get {...} }

      public bool IsError { get {...} }
   }
}
```

Если программа использует тип или элемент, к которому применен атрибут `Obsolete`, компилятор выводит предупреждение или ошибку. Точнее, если параметр `error` не указан или указан, но имеет значение `false`, компилятор выводит предупреждение. Если параметр `error` указан и имеет значение `true`, компилятор выводит ошибку.

Пример:

```
[Obsolete("Этот класс является устаревшим; используйте вместо него класс B")]

class A
{
    public void F() { }
}

class B
{
    public void F() { }
}

class Test
{
    static void Main()
   {
      A a = new A();            // Предупреждение
      a.F();
   }
}
```

Здесь к классу `A` применен атрибут `Obsolete`. Каждое использование класса `A` в `Main` приведет к предупреждению, включающему в себя указанное сообщение, `Этот класс является устаревшим; используйте вместо него класс B`.

> **ДЖОН СКИТ**
>
> В этом примере содержится важный урок: когда вы делаете что-то устаревшим, всегда давайте указания касательно нового, более предпочтительного способа достижения нужного эффекта. В идеальном варианте также предоставьте какую-нибудь подсказку (возможно, в документации, а не в сообщении атрибута) о том, почему «старый» код

был объявлен «устаревшим». Невероятно раздражает, когда код, который долгое время работал, вдруг начинает без достаточных объяснений сыпать предупреждениями.

МАРЕК САФАР

Компилятор автоматически не делает устаревшими переопределения виртуальных методов. Таким образом, если виртуальный метод отмечен как `Obsolete`, к каждому методу, который его переопределяет, также должен быть вручную применен атрибут `Obsolete`.

17.5. Атрибуты для взаимодействия

Примечание. Этот раздел применим только к реализации C# на платформе .NET.

17.5.1. Взаимодействие с компонентами COM и Win32

Среда выполнения .NET предоставляет большое количество атрибутов, позволяющих программам на C# взаимодействовать с компонентами, написанными с использованием COM и библиотек динамической компоновки (DLL) Win32. Например, атрибут `DllImport` может использоваться с методом, объявленным как `static extern`, для указания на то, что реализацию этого метода следует искать в Win32 DLL. Эти атрибуты находятся в пространстве имен `System.Runtime.InteropServices`, а детальную документацию по ним можно найти в документации к среде выполнения .NET.

17.5.2. Взаимодействие с другими языками .NET

17.5.2.1. Атрибут IndexerName

Индексаторы в .NET реализованы с использованием индексированных свойств и имеют имя в метаданных .NET. Если для индексатора не представлен атрибут `IndexerName`, по умолчанию используется имя `Item`. Атрибут `IndexerName` позволяет разработчику переопределить это умолчание и указать другое имя.

```
namespace System.Runtime.CompilerServices.CSharp
{
   [AttributeUsage(AttributeTargets.Property)]
   public class IndexerNameAttribute : Attribute
   {
      public IndexerNameAttribute(string indexerName) {...}

      public string Value { get {...} }
   }
}
```

Глава 18
Небезопасный код

Основная часть языка C#, описанная в предыдущих главах, существенно отличается от C и C++ отсутствием указателей как типов данных. Вместо них в C# присутствуют ссылки и возможность создавать объекты, управляемые сборщиком мусора. Такое проектное решение вместе с другими возможностями делает C# намного более безопасным языком, чем C или C++. В основной части C# попросту невозможно создать неинициализированную переменную, «висячий» указатель или выражение, обращающееся к массиву по индексу, выходящему за его пределы. Таким образом, устраняются целые категории ошибок, регулярно появляющихся в программах на C и C++.

Несмотря на то что практически каждая конструкция с типом указателя в C и C++ имеет свой аналог ссылочного типа в C#, все равно существуют ситуации, в которых необходим доступ к типам указателей. Например, без доступа к указателям становятся невозможными или неудобными такие вещи, как взаимодействие с операционной системой, доступ к отображаемому в память устройству или реализация критического с точки зрения времени алгоритма. Чтобы удовлетворить эту потребность, в C# существует возможность писать **небезопасный код**.

> **КРИС СЕЛЛЗ**
>
> За все время, что я занимаюсь программированием с использованием .NET (еще до выхода версии 1.0), мне ни разу не потребовалось писать небезопасный код. Ни разу.

В небезопасном коде можно объявлять и использовать указатели, осуществлять преобразования между указателями и целочисленными типами, получать адреса переменных и т. д. В известной степени написание небезопасного кода схоже с написанием кода на C внутри программы на C#.

На самом деле небезопасный код является «безопасной» возможностью с точки зрения как разработчиков, так и пользователей. Небезопасный код должен быть явно отмечен модификатором `unsafe`, поэтому разработчики не могут случайно воспользоваться небезопасными возможностями, а механизм выполнения гарантирует, что небезопасный код не может выполняться в ненадежной среде.

18.1. Небезопасные контексты

Небезопасные возможности C# доступны только в *небезопасных контекстах*. Небезопасный контекст создается путем добавления модификатора `unsafe` в объявление типа или элемента или с помощью *оператора-unsafe*:

- Объявление класса, структуры, интерфейса или делегата может содержать модификатор `unsafe`, при этом все текстовое пространство данного объявления типа (включая тело класса, структуры или интерфейса) рассматривается как небезопасный контекст.
- Объявление поля, метода, свойства, события, индексатора, операции, конструктора экземпляра, деструктора или статического конструктора может содержать модификатор `unsafe`, при этом все текстовое пространство объявления данного элемента рассматривается как небезопасный контекст.
- *Оператор-unsafe* позволяет использовать небезопасный контекст внутри *блока*. Все текстовое пространство ассоциированного с оператором *блока* рассматривается как небезопасный контекст.

Ниже приведены соответствующие дополнения правил грамматики. Правила, описанные в предыдущих частях, для краткости заменены многоточиями (...).

```
модификатор-класса:
    …
    unsafe

модификатор-структуры:
    …
    unsafe

модификатор-интерфейса:
    …
    unsafe

модификатор-делегата:
    …
    unsafe

модификатор-поля:
    …
    unsafe

модификатор-метода:
    …
    unsafe

модификатор-свойства:
    …
    unsafe

модификатор-события:
    …
    unsafe
```

модификатор-индексатора:
…
unsafe

модификатор-операции:
…
unsafe

модификатор-конструктора:
…
unsafe

объявление-деструктора:
атрибуты$_{opt}$ **extern**$_{opt}$ **unsafe**$_{opt}$ **~** *идентификатор* **(** **)** *тело-деструктора*
атрибуты$_{opt}$ **unsafe**$_{opt}$ **extern**$_{opt}$ **~** *идентификатор* **(** **)** *тело-деструктора*

модификаторы-статического-конструктора:
extern$_{opt}$ **unsafe**$_{opt}$ **static**
unsafe$_{opt}$ **extern**$_{opt}$ **static**
extern$_{opt}$ **static** **unsafe**$_{opt}$
unsafe$_{opt}$ **static** **extern**$_{opt}$
static **extern**$_{opt}$ **unsafe**$_{opt}$
static **unsafe**$_{opt}$ **extern**$_{opt}$

вложенный-оператор:
…
оператор-unsafe

оператор-unsafe:
unsafe *блок*

Пример:

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

Здесь модификатор `unsafe`, указанный в объявлении структуры, делает все текстовое пространство объявления структуры небезопасным контекстом. В результате становится возможным объявлять поля `Left` и `Right`, имеющие тип указателя. Этот пример можно также переписать следующим образом:

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Здесь модификаторы `unsafe` в объявлениях полей приводят к тому, что эти объявления рассматриваются как небезопасные контексты.

Модификатор `unsafe` не оказывает никакого другого влияния на тип или элемент, кроме того, что он создает небезопасный контекст, разрешая тем самым использование типов указателей.

Пример:

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}

public class B : A
{
    public override void F() {
        base.F();
        ...
    }
}
```

В этом примере использование модификатора `unsafe` в методе `F` класса `A` просто приводит к тому, что текстовое пространство `F` становится небезопасным контекстом, в котором могут использоваться небезопасные возможности языка. Нет необходимости повторно использовать модификатор `unsafe` в переопределенном методе `F` в классе `B` — разумеется, до тех пор, пока в этом методе тоже не потребуется доступ к небезопасным возможностям.

Немного отличается ситуация, в которой тип указателя является частью сигнатуры метода:

```
public unsafe class A
{
    public virtual void F(char* p) {...}
}

public class B : A
{
    public unsafe override void F(char* p) {...}
}
```

Здесь метод `F` может быть записан только в небезопасном контексте, так как его сигнатура содержит тип указателя. Однако такой контекст можно создать или сделав весь класс небезопасным, как в случае класса `A`, или же использовав модификатор `unsafe` в объявлении метода, как это сделано в классе `B`.

18.2. Типы указателей

В небезопасном контексте *тип* (раздел 4) может быть *типом-указателя*, а также *типом-значением* или *ссылочным-типом*. Однако *тип-указателя* также можно использовать в выражении `typeof` (раздел 7.6.10.6) за пределами небезопасного контекста, поскольку подобное использование не является небезопасным.

тип:
    …
    тип-указателя

Тип-указателя записывается как *неуправляемый-тип* или ключевое слово `void`, за которым следует лексема `*`:

```
тип-указателя:
    неуправляемый-тип   *
    void  *

неуправляемый-тип:
    тип
```

Тип, записанный перед символом `*` в типе указателя, называется **указуемым типом** типа указателя. Он представляет собой тип переменной, на которую указывает значение типа указателя.

В отличие от ссылок (значений ссылочных типов) указатели не отслеживаются сборщиком мусора — у него нет сведений об указателях и о данных, на которые они указывают. По этой причине указатель не может указывать на ссылку или на структуру, содержащую ссылки, а указуемый тип указателя должен являться *неуправляемым-типом*.

Неуправляемый-тип — это любой тип, не являющийся *ссылочным-типом* или сконструированным типом и не содержащий полей *ссылочного-типа* или сконструированного типа на любом уровне вложенности. Иными словами, *неуправляемый-тип* — это один из следующих типов:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` или `bool`.
- Любой *перечислимый-тип*.
- Любой *тип-указателя*.
- Любой определенный пользователем *структурный-тип*, не являющийся сконструированным типом и содержащий только поля, имеющие *неуправляемые-типы*.

Интуитивно понятное правило сочетания указателей и ссылок заключается в следующем: указуемые объекты ссылок могут содержать указатели, но указуемые объекты указателей не могут содержать ссылок.

Некоторые примеры типов указателей приведены в таблице.

Пример	**Описание**
`byte*`	Указатель на `byte`
`char*`	Указатель на `char`
`int**`	Указатель на указатель на `int`
`int*[]`	Одномерный массив указателей на `int`
`void*`	Указатель на неизвестный тип

Все типы указателей для конкретной реализации должны иметь один и тот же размер и представление.

В отличие от C и C++, когда в одном объявлении объявляются несколько указателей, в C# символ * записывается только рядом с типом, а не в качестве префикса для каждого имени указателя. Пример:

```
int* pi, pj;        // НЕ int *pi, *pj;
```

Значение указателя, имеющего тип `T*`, представляет собой *адрес* переменной типа `T`. Для доступа к этой переменной можно использовать операцию разыменования указателя * (раздел 18.5.1). Например, при наличии переменной `P` типа `int*` выражение `*P` обозначает переменную типа `int`, расположенную по адресу, содержащемуся в `P`.

Как и ссылка на объект, указатель может иметь значение `null`. Результат применения операции разыменования к такому указателю зависит от реализации. Во всех битах указателя со значением `null` содержатся нули.

Тип `void*` представляет собой указатель на неизвестный тип. Поскольку указуемый тип неизвестен, к такому указателю нельзя применить операцию разыменования, а также произвести с ним какие-либо арифметические действия. Однако указатель типа `void*` можно привести к любому типу указателя (и наоборот).

Типы указателей составляют отдельную категорию типов. В отличие от ссылочных типов и типов-значений, они не наследуются от `object`, и не существует преобразований между типами указателей и `object`. В частности, для указателей не поддерживается упаковка и распаковка (раздел 4.3). Однако допустимы преобразования между различными типами указателей, а также между типами указателей и целочисленными типами. Об этом написано в разделе 18.4.

Тип-указателя не может быть использован в качестве аргумента типа (раздел 4.4), а выведение типов (раздел 7.5.2) не работает при вызовах обобщенных методов, для которых выведенный аргумент типа может оказаться типом указателя.

Тип-указателя можно использовать в качестве типа асинхронно-изменяемого поля (раздел 10.5.3).

Несмотря на то что указатели допускается передавать как параметры `ref` или `out`, это может привести к неопределенному поведению, поскольку указателю может быть присвоен адрес локальной переменной, которая перестанет существовать после возврата из метода, или если фиксированный объект, на который указывал указатель, перестает быть фиксированным. Пример:

```
using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2)
    {
        int i = 10;
        pi1 = &i;
        fixed (int* pj = &value)
        {
            // ...
            pi2 = pj;
        }
    }
```

```
    static void Main()
    {
        int i = 10;
        unsafe
        {
            int* px1;
            int* px2 = &i;
            F(out px1, ref px2);
            Console.WriteLine("*px1 = {0}, *px2 = {1}", *px1, *px2);
            // Неопределенное поведение
        }
    }
}
```

Метод может возвращать значение какого-либо типа, и этот тип может быть указателем. Например, следующий метод, если передать ему указатель на непрерывную последовательность `int`, число элементов в этой последовательности и какое-то другое значение `int`, возвращает адрес этого значения в данной последовательности, если оно найдено, а в противном случае возвращает `null`:

```
unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
                return pi;
        ++pi;
    }
    return null;
}
```

В небезопасном контексте для работы с указателями доступны несколько конструкций:

- Операция `*` может быть использована для разыменования указателей (раздел 18.5.1).
- Операция `->` может быть использована для доступа к элементу структуры через указатель (раздел 18.5.2).
- Операция `[]` может быть использована для доступа к указателю по индексу (раздел 18.5.3).
- Операция `&` может быть использована для получения адреса переменной (раздел 18.5.4).
- Операции `++` и `--` могут быть использованы для инкремента и декремента указателей (раздел 18.5.5).
- Операции `+` и `-` могут быть использованы для выполнения арифметических действий с указателями (раздел 18.5.6).
- Операции `==`, `!=`, `<`, `>`, `<=` и `>=` могут быть использованы для сравнения указателей (раздел 18.5.7).
- Операция `stackalloc` может быть использована для выделения памяти из стека вызовов (раздел 18.7).
- Оператор `fixed` может быть использован для временной фиксации переменной, с тем чтобы можно было получить ее адрес (раздел 18.6).

18.3. Фиксированные и перемещаемые переменные

Операция получения адреса (раздел 18.5.4) и оператор `fixed` (раздел 18.6) делят переменные на две категории: **фиксированные переменные** и **перемещаемые переменные**.

Фиксированные переменные хранятся в тех областях памяти, которые не затрагиваются при работе сборщика мусора. (Примерами таких переменных являются локальные переменные, параметры-значения и переменные, созданные разыменованием указателей.) Перемещаемые переменные, напротив, располагаются в тех областях памяти, которые могут быть перемещены или уничтожены сборщиком мусора. (Примерами таких переменных являются поля объектов и элементы массивов.)

Операция `&` (раздел 18.5.4) позволяет без ограничений получить адрес фиксированной переменной. Однако адрес перемещаемой переменной можно получить только с помощью оператора `fixed` (раздел 18.6), поскольку она может быть перемещена или уничтожена сборщиком мусора, и этот адрес остается действительным только на время действия оператора `fixed`.

Говоря точнее, фиксированной является одна из следующих переменных:

- Переменная, получаемая из *простого-имени* (раздел 7.6.2), которое ссылается на локальную переменную или параметр-значение, если только эта переменная не «захвачена» анонимной функцией.
- Переменная, получаемая с помощью *доступа-к-элементу* (раздел 7.6.4) в форме `V.I`, где `V` — фиксированная переменная *структурного-типа*.
- Переменная, получаемая из *выражения-разыменования-указателя* (раздел 18.5.1) в форме `*P`, *доступа-к-элементу-объекта-через-указатель* (раздел 18.5.2) в форме `P->I` или *доступа-к-элементу-через-указатель* (раздел 18.5.3) в форме `P[E]`.

Все остальные переменные классифицируются как перемещаемые.

Заметьте, что статическое поле классифицируется как перемещаемая переменная. Также заметьте, что параметр `ref` или `out` классифицируется как перемещаемая переменная, даже если аргумент, переданный в качестве параметра, является фиксированной переменной. И наконец, заметьте, что переменная, получаемая в результате разыменования указателя, всегда классифицируется как фиксированная.

18.4. Преобразования указателей

В небезопасном контексте набор доступных неявных преобразований (раздел 6.1) дополняется следующими неявными преобразованиями указателей:

- Из любого *типа-указателя* в тип `void*`.
- Из литерала `null` в любой *тип-указателя*.

Кроме этого, в небезопасном контексте набор доступных явных преобразований (раздел 6.2) дополняется следующими явными преобразованиями указателей:

- Из любого *типа-указателя* в любой другой *тип-указателя*.
- Из `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong` в любой *тип-указателя*.
- Из любого *типа-указателя* в `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`.

И наконец, в небезопасном контексте набор стандартных неявных преобразований (раздел 6.3.1) включает в себя следующее преобразование указателя:

- Из любого *типа-указателя* в тип `void*`.

Преобразования между двумя типами указателей никогда не меняют фактического значения указателя. Иными словами, преобразование из одного типа указателя в другой не оказывает влияния на адрес, хранящийся в указателе.

Когда один тип указателя преобразуется в другой и при этом результирующий указатель не выровнен в соответствии с типом, на который он указывает, поведение при разыменовании результата не определено. В общем случае понятие «корректно выровнен» транзитивно: если указатель на тип `A` корректно выровнен для указателя на тип `B`, который, в свою очередь, корректно выровнен для указателя на тип `C`, то указатель типа `A` корректно выровнен для указателя на тип `C`.

Рассмотрим ситуацию, в которой доступ к переменной одного типа осуществляется через указатель на переменную другого типа:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;            // Не определено
*pi = 123456;           // Не определено
```

Когда выполняется преобразование типа указателя в указатель на `byte`, результат указывает на младший адресуемый байт переменной. Последующие операции инкремента результата, вплоть до размера переменной, в качестве результата будут возвращать указатели на остальные байты этой переменной. Например, следующий метод отображает каждый из восьми байт переменной типа `double` в виде шестнадцатеричного значения:

```
using System;
class Test
{
    unsafe static void Main()
    {
        double d = 123.456e23;
        unsafe
        {
            byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
            Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

Разумеется, вывод зависит от порядка следования байтов.

> **ПИТЕР СЕСТОФТ**
>
> Одним из вариантов использования небезопасного кода является выполнение преобразований, основанных на иной интерпретации последовательностей бит, представляющих простейшие данные, структуры или ссылки, используя небезопасное приведение типов указателей. В большинстве случаев результат будет непереносимым или бессмысленным. Однако иная интерпретация может быть использована для того, чтобы переносимым способом получать и устанавливать значения бит полезной нагрузки NaN в числах с плавающей точкой в формате IEEE (см. комментарий в разделе 7.8.1):

```
unsafe static int GetNaNPayload(float f) {
    float* p = &f;
    return *((int*)p) & 0x003FFFFF;
}

unsafe static float MakeNaNPayload(int nanbits) {
    float nan = Single.NaN;
    float* p = &nan;
    *((int*)p) |= (nanbits & 0x003FFFFF);
    return nan;
}
```

> Небезопасный код может и не быть столь непонятным, но он, вероятнее всего, *будет* неправильным, потому что стандартные проверки на осмысленность не выполняются.

Сопоставление указателей и целых чисел зависит от реализации. Однако на 32- и 64-битных процессорных архитектурах с линейным адресным пространством преобразование указателей в целочисленные типы и обратно обычно работает так же, как преобразование значений типа `uint` или `ulong` в эти целочисленные типы и обратно.

18.4.1. Массивы указателей

В небезопасном контексте можно создавать массивы указателей. Для таких массивов допустима только часть преобразований, применяемых к другим типам массивов:

- Неявное преобразование ссылки (раздел 6.1.6) из любого *типа-массива* в класс `System.Array` и реализуемые им интерфейсы также применимы и к массивам указателей. Однако любая попытка доступа к элементам массива через `System.Array` или реализуемые им интерфейсы приведет к возникновению исключения во время выполнения, так как типы указателей нельзя преобразовать в `object`.
- К массивам указателей неприменимы неявные и явные преобразования ссылок (разделы 6.1.6, 6.2.4) из типа одномерного массива `S[]` в `System.Collections.Generic.IList<T>` и его базовые интерфейсы, так как типы указателей не могут использоваться в качестве аргументов типа, и в данном случае не существует преобразований из типов указателей в типы, не являющиеся указателями.

- Явное преобразование ссылки (раздел 6.2.4) из `System.Array` и реализуемых им интерфейсов в любой *тип-массива* применимо к массивам указателей.
- К массивам указателей неприменимы явные преобразования ссылок (раздел 6.2.4.) из `System.Collections.Generic.IList<S>` и его базовых интерфейсов в тип одномерного массива `T[]`, так как типы указателей не могут использоваться в качестве аргументов типа, и в данном случае не существует преобразований типов указателей в типы, не являющиеся указателями.

Эти ограничения означают, что распространение оператора `foreach` на массивы, описанное в разделе 8.8.4, неприменимо к массивам указателей. Вместо этого оператор `foreach` в форме

```
foreach (V v in x) вложенный-оператор
```

где `x` имеет тип массива в форме `T[,,…,]`, *n* — число измерений массива минус 1, а `T` или `V` имеют тип указателя, разворачивается с помощью вложенных циклов `for` следующим образом:

```
{
    T[,,...,] a = x;
    V v;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int in = a.GetLowerBound(n); in <= a.GetUpperBound(n); in++) {
        v = (V)a.GetValue(i0,i1,...,in);
        вложенный-оператор
    }
}
```

Переменные `a`, `i0`, `i1`, ..., `in` невидимы и недоступны для `x`, *вложенного-оператора* или любого другого исходного кода программы. Переменная `v` доступна только для чтения во *вложенном-операторе*. Если не существует явного преобразования (раздел 18.4) из `T` (тип элемента) в `V`, возникает ошибка и дальнейшие действия не выполняются. Если `x` имеет значение `null`, во время выполнения выбрасывается исключение `System.NullReferenceException`.

18.5. Использование указателей в выражениях

В небезопасном контексте результат выражения может иметь тип указателя. Если выражение имеет тип указателя за пределами небезопасного контекста, возникает ошибка компиляции. Точнее говоря, за пределами безопасного контекста ошибка компиляции возникает, если любое *простое-имя* (раздел 7.6.2), результат *доступа-к-элементу-объекта* (раздел 7.6.4), вычисления *выражения-вызова* (раздел 7.6.5) или *доступа-к-элементу* (раздел 7.6.6) имеет тип указателя.

В небезопасном контексте грамматические правила для *первичных-выражений-не-создающих-массив* (раздел 7.6) и *унарных-выражений* (раздел 7.7) допускают следующие дополнительные конструкции:

```
первичное-выражение-без-создания-массива:
    …
    доступ-к-элементу-объекта-через-указатель
    доступ-к-элементу-через-указатель
    выражение-sizeof

унарное-выражение:
    …
    выражение-разыменования-указателя
    выражение-получения-адреса
```

Эти конструкции описаны в следующих разделах. Приоритеты и ассоциативность небезопасных операций подразумеваются грамматикой.

18.5.1. Разыменование указателей

Выражение-разыменования-указателя состоит из звездочки (*), за которой следует *унарное-выражение*.

```
выражение-разыменования-указателя:
    *    унарное-выражение
```

Унарная операция * обозначает *разыменование указателя* и используется для получения переменной, на которую он указывает. Результатом вычисления `*P`, где `P` — выражение типа указателя `T*`, является переменная типа `T`. Применение унарной операции * к выражению типа `void*` или выражению, тип которого отличается от указателя, приводит к ошибке компиляции.

Результат применения унарной операции * к указателю, имеющему значение `null`, зависит от реализации. В частности, нет гарантии того, что эта операция выбросит `System.NullReferenceException`.

Если указателю было присвоено некорректное значение, поведение унарной операции * не определено. К некорректным значениям при разыменовании указателя этой операцией относятся адреса, некорректно выровненные для типов, на которые они указывают (см. пример в разделе 18.4), а также адрес переменной, у которой истекло время жизни.

При проверке того, является ли переменная явно присвоенной, переменная, полученная в результате вычисления выражения в форме `*P`, считается инициализированной (раздел 5.3.1).

18.5.2. Доступ к элементу объекта через указатель

Доступ-к-элементу-объекта-через-указатель состоит из *первичного-выражения*, за которым следуют лексема `->` и *идентификатор*:

```
доступ-к-элементу-объекта-через-указатель:
    первичное-выражение    ->    идентификатор    список-аргументов-типа_opt
```

В доступе к элементу объекта через указатель в форме `P->I` `P` должно являться выражением типа указателя, отличного от `void*`, а `I` должно обозначать доступный элемент типа, на который указывает `P`.

> **ВЛАДИМИР РЕШЕТНИКОВ**
>
> Вы не можете применить *доступ-к-элементу-объекта-через-указатель* к типам, представляющим собой указатель на указатель (например, `int**`), так как указатели не содержат элементов.

Доступ к элементу объекта через указатель в форме `P->I` вычисляется в точности так же, как `(*P).I`. Описание операции разыменования указателя `(*)` приведено в разделе 18.5.1. Описание операции доступа к элементу объекта `(.)` приведено в разделе 7.6.4.

Пример:

```
using System;
struct Point
{
    public int x;
    public int y;
    public override string ToString()
    {
        return "(" + x + "," + y + ")";
    }
}
class Test
{
    static void Main()
    {
        Point point;
        unsafe
        {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

В этом примере оператор `->` используется для доступа к полям и вызова метода структуры через указатель. Так как действие `P->I` в точности эквивалентно `(*P).I`, метод `Main` можно было записать и следующим образом:

```
class Test
{
    static void Main()
    {
        Point point;
        unsafe
        {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

18.5.3. Доступ к элементу через указатель

Доступ-к-элементу-через-указатель состоит из *первичного-выражения-не-создающего-массив*, за которым следует выражение, заключенное в квадратные скобки «[» и «]».

```
доступ-к-элементу-через-указатель:
    первичное-выражение-без-создания-массива   [   выражение   ]
```

В доступе к элементу через указатель в форме `P[E]` `P` должно являться выражением типа указателя, отличного от `void*`, а `E` должно быть выражением, которое можно неявно преобразовать в `int`, `uint`, `long` или `ulong`.

Доступ к элементу через указатель в форме `P[E]` вычисляется в точности так же, как `*(P+E)`. Описание операции разыменования указателя (`*`) приведено в разделе 18.5.1. Описание операции сложения указателей (`+`) приведено в разделе 18.5.6.

Пример:

```
class Test
{
    static void Main()
    {
        unsafe
        {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

Здесь доступ к элементу через указатель используется для инициализации буфера символов в цикле `for`. Так как действие `P[E]` в точности эквивалентно `*(P+E)`, данный пример можно было записать и следующим образом:

```
class Test
{
    static void Main()
    {
        unsafe
        {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

Операция доступа к элементу через указатель не выполняет проверку выхода за границы, а поведение при доступе к элементу за пределами массива не определено. Такой же подход используется в C и C++.

18.5.4. Операция получения адреса

Выражение-получения-адреса состоит из амперсанда (`&`), за которым следует *унарное-выражение*.

выражение-получения-адреса:
    &   *унарное-выражение*

Для выражения `E`, имеющего тип `T` и классифицируемого как фиксированная переменная (раздел 18.3), конструкция `&E` вычисляет адрес переменной, заданной этим выражением. Типом результата является `T*`, и он классифицируется как значение. Если `E` не классифицируется как переменная или как локальная переменная, доступная только для чтения, или обозначает перемещаемую переменную, возникает ошибка компиляции. В последнем случае можно использовать оператор фиксации (раздел 18.6), чтобы временно «зафиксировать» переменную, перед тем как получить ее адрес. Как говорилось в разделе 7.6.4, за пределами конструктора экземпляра или статического конструктора структуры или класса, определяющих доступное только для чтения (`readonly`) поле, это поле считается значением, а не переменной. Получить его адрес как таковой невозможно. Также нельзя получить адрес константы.

Для выполнения операции `&` не требуется, чтобы ее аргумент был явно присвоен, но после выполнения этой операции переменная, к которой она применялась, считается явно присвоенной на всем пути выполнения, в котором встретилась операция. Ответственность за проведение корректной инициализации переменной в данном случае ложится на программиста.

Пример:

```
using System;

class Test
{
    static void Main()
    {
        int i;
        unsafe
        {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

Здесь `i` считается явно присвоенной после того, как для инициализации `p` была использована операция `&i`. Выполнение присваивания `*p` на самом деле инициализирует `i`, но ответственность за то, что эта инициализация будет присутствовать, лежит на программисте, и в случае удаления присваивания ошибка компиляции не возникнет.

Существуют правила явного присваивания для операции `&`, позволяющие избежать избыточной инициализации локальных переменных. Например, многие внешние API принимают указатели на структуры, которые они инициализируют самостоятельно. При осуществлении вызовов таких API им обычно передается адрес локальной переменной типа структуры, и без этих правил могла бы потребоваться избыточная инициализация данной переменной.

18.5.5. Инкремент и декремент указателей

В небезопасном контексте операции `++` и `--` (разделы 7.6.9 и 7.7.5) могут применяться к переменным, представляющим собой указатели любых типов за исключением `void*`. Таким образом, для каждого типа указателя `T*` неявно определены следующие операции:

```
T* operator ++(T* x);
T* operator --(T* x);
```

Результаты этих операций аналогичны результатам `x + 1` и `x – 1` соответственно (раздел 18.5.6). Иными словами, для переменной, являющейся указателем типа `T*`, операция `++` прибавляет `sizeof(T)` к адресу, хранящемуся в переменной, а операция `--` вычитает `sizeof(T)` из этого адреса.

Если операция инкремента или декремента приводит к переполнению типа указателя, результат зависит от реализации, но никаких исключений не возникает.

18.5.6. Арифметика указателей

В небезопасном контексте операции `+` и `-` (разделы 7.8.4 и 7.8.5) могут применяться к значениям указателей всех типов за исключением `void*`. Таким образом, для каждого типа указателя `T*` неявно определены следующие операции:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

Для выражения `P`, имеющего тип указателя `T*`, и выражения `N`, имеющего тип `int`, `uint`, `long` или `ulong`, выражения `P + N` и `N + P` вычисляют значение указателя типа `T*`, получаемое путем прибавления `N * sizeof(T)` к адресу, заданному `P`. Аналогично, выражение `P - N` вычисляет значение указателя типа `T*`, получаемое путем вычитания `N * sizeof(T)` из адреса, заданного `P`.

Для двух выражений `P` и `Q`, имеющих тип указателя `T*`, выражение `P - Q` вычисляет разность адресов, заданных `P` и `Q`, а затем делит эту разность на `sizeof(T)`. Результат всегда имеет тип `long`. Фактически, `P - Q` вычисляется как `((long)(P) - (long)(Q)) / sizeof(T)`. Пример:

```
using System;

class Test
{
    static void Main()
    {
        unsafe
        {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

Эта программа выводит:

```
p - q = -14
q - p = 14
```

Если при выполнении арифметической операции с указателем происходит переполнение типа указателя, результат усекается способом, зависящим от реализации, но никаких исключений не возникает.

18.5.7. Сравнение указателей

В небезопасном контексте операции `==`, `!=`, `<`, `>`, `<=` и `>=` (раздел 7.10) могут применяться к значениям указателей всех типов. Операции сравнения указателей определены следующим образом:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Так как существует неявное преобразование из указателя любого типа в тип `void*`, с помощью этих операций можно сравнивать операнды любого типа указателей. Операции сравнения сравнивают адреса, заданные двумя операндами, как если бы они были беззнаковыми целыми числами.

18.5.8. Операция sizeof

Операция `sizeof` возвращает количество байт, занимаемых переменной заданного типа. Тип, указанный в качестве операнда для `sizeof`, должен быть *неуправляемым-типом* (раздел 18.2).

выражение-sizeof:
    sizeof **(** *неуправляемый-тип* **)**

Результатом операции `sizeof` является значение типа `int`. Для некоторых встроенных типов результатом операции `sizeof` является константное значение в соответствии с приведенной далее таблицей.

Выражение	**Результат**
`sizeof(sbyte)`	`1`
`sizeof(byte)`	`1`
`sizeof(short)`	`2`
`sizeof(ushort)`	`2`
`sizeof(int)`	`4`
`sizeof(uint)`	`4`
`sizeof(long)`	`8`
`sizeof(ulong)`	`8`
`sizeof(char)`	`2`
`sizeof(float)`	`4`
`sizeof(double)`	`8`
`sizeof(bool)`	`1`

Для всех остальных типов результат операции `sizeof` зависит от реализации и классифицируется как значение, а не как константа.

Порядок, в котором располагаются элементы структуры, не определен.

В целях выравнивания в начале, в конце или внутри структуры могут присутствовать безымянные заполнения (padding). Значения бит, используемых для заполнения, не определены.

Когда операция `sizeof` применяется к операнду структурного типа, результатом является общее количество байт в переменной этого типа, включая любые заполнения.

18.6. Оператор fixed

В небезопасном контексте грамматические правила для *вложенных-операторов* (раздел 8) допускают использование дополнительной конструкции — оператора `fixed`, который используется для «фиксации» перемещаемой переменной, с тем чтобы ее адрес оставался неизменным на время действия оператора.

вложенный-оператор:
  …
  оператор-fixed

оператор-fixed:
  fixed **(** *тип-указателя* *описатели-фиксированного-указателя* **)**
    вложенный-оператор

описатели-фиксированного-указателя:
  описатель-фиксированного-указателя
  описатели-фиксированного-указателя **,** *описатель-фиксированного-указателя*

```
описатель-фиксированного-указателя:
    идентификатор   =   инициализатор-фиксированного-указателя

инициализатор-фиксированного-указателя:
    &   ссылка-на-переменную
    выражение
```

Каждый *описатель-фиксированного-указателя* объявляет локальную переменную заданного *типа-указателя* и инициализирует ее адресом, вычисленным соответствующим *инициализатором-фиксированного-указателя*. Локальная переменная, объявленная в операторе `fixed`, доступна для любых *инициализаторов-фиксированного-указателя*, находящихся справа от объявления этой переменной, а также во *вложенном-операторе* оператора `fixed`. Локальная переменная, объявленная в операторе `fixed`, доступна только для чтения. При попытке модифицировать ее во вложенном операторе (путем присваивания или использования операций `++` и `--`) или передать ее в качестве параметра `ref` или `out` возникает ошибка компиляции.

В роли *инициализатора-фиксированного-указателя* может выступать одна из следующих сущностей:

- Лексема `&`, за которой следует *ссылка-на-переменную* (раздел 5.3.3), ссылающаяся на перемещаемую переменную (раздел 18.3) неуправляемого типа `T`, при условии, что тип `T*` может быть неявно преобразован в тип указателя, заданный в операторе `fixed`. В этом случае инициализатор вычисляет адрес заданной переменной, и гарантируется, что этот адрес будет оставаться неизменным во время действия оператора `fixed`.
- Выражение *типа-массива* с элементами неуправляемого типа `T`, при условии, что тип `T*` может быть неявно преобразован в тип указателя, заданный в операторе `fixed`. В этом случае инициализатор вычисляет адрес первого элемента массива, и гарантируется, что весь массив будет располагаться по фиксированному адресу во время действия оператора `fixed`. Если выражение типа массива имеет значение `null` или массив пуст, поведение оператора `fixed` зависит от реализации.
- Выражение типа `string`, при условии, что тип `char*` может быть неявно преобразован в тип указателя, заданный в операторе `fixed`. В этом случае инициализатор вычисляет адрес первого символа строки, и гарантируется, что вся строка будет располагаться по фиксированному адресу во время действия оператора `fixed`. Если выражение строкового типа имеет значение `null`, поведение оператора `fixed` зависит от реализации.
- *Простое-имя* или *доступ-к-элементу-объекта*, ссылающиеся на элемент, являющийся буфером фиксированного размера, расположенный в перемещаемой переменной, при условии, что тип этого элемента может быть неявно преобразован в тип указателя, заданный в операторе `fixed`. В этом случае инициализатор вычисляет указатель на первый элемент буфера фиксированного размера (раздел 18.7.2), и гарантируется, что этот буфер будет располагаться по фиксированному адресу во время действия оператора `fixed`.

> **ЭРИК ЛИППЕРТ**
>
> В этом пункте продемонстрировано объединение понятий, которое потенциально может сбивать с толку: ключевое слово `fixed` используется как в значении «фиксированное расположение», так и в значении «фиксированный размер». Путаницу увеличивает тот факт, что элемент массива фиксированного размера должен иметь фиксированное расположение, чтобы его можно было использовать. Постарайтесь запомнить, что блок фиксированного размера не всегда имеет фиксированное расположение.

Для каждого адреса, вычисленного *инициализатором-фиксированного-указателя*, оператор `fixed` гарантирует, что переменная, расположенная по этому адресу, не будет перемещена или уничтожена сборщиком мусора во время действия оператора. Например, если адрес, вычисленный *инициализатором-фиксированного-указателя*, ссылается на поле в объекте или на элемент экземпляра массива, оператор `fixed` гарантирует, что экземпляр объекта не будет перемещен или уничтожен во время действия оператора.

Ответственность за то, что указатели, созданные операторами `fixed`, не будут использоваться за пределами этих операторов, лежит на программисте. Например, когда указатели, созданные операторами `fixed`, передаются во внешние API, программист должен удостовериться, что API не сохраняют эти указатели.

> **ЭРИК ЛИППЕРТ**
>
> Если возникла неудачная ситуация, в которой вам требуется зафиксировать расположение блока управляемой памяти на более длительное время, чем действует оператор `fixed`, вы можете сделать это, используя тип `GCHandle`. Тем не менее лучше все-таки избегать подобных ситуаций.

Фиксированные объекты могут приводить к фрагментации кучи (поскольку их нельзя перемещать). По этой причине объекты лучше фиксировать, только когда это совершенно необходимо и только на максимально короткие промежутки времени. Пример:

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p)
    {
        *p = 1;
    }

    static void Main()
    {
        Test t = new Test();
        int[] a = new int[10];
        unsafe
        {
```

```
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

В этом примере продемонстрировано несколько вариантов использования оператора `fixed`. Первый оператор фиксирует и получает адрес статического поля, второй фиксирует и получает адрес поля экземпляра, а третий фиксирует и получает адрес элемента массива. В каждом случае использование обычной операции `&` привело бы к ошибке, так как все переменные классифицируются как перемещаемые.

Четвертый оператор `fixed` в этом примере приводит к такому же результату, что и третий.

В следующем примере в операторе `fixed` используется тип `string`:

```
class Test
{
    static string name = "xx";

    unsafe static void F(char* p)
    {
        for (int i = 0; p[i] != ‘\0’; ++i)
        Console.WriteLine(p[i]);
    }

    static void Main()
    {
        unsafe
        {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```

В небезопасном контексте элементы одномерных массивов хранятся в порядке увеличения индекса, начиная с индекса `0` и заканчивая индексом `Length - 1`. Элементы многомерных массивов хранятся таким образом, что сначала увеличиваются индексы крайнего правого измерения, затем стоящего слева от него и так далее, справа налево. Внутри оператора `fixed`, который получает указатель `p` на экземпляр массива `a`, значения указателей, лежащие в диапазоне от `p` до `p + a.Length - 1`, представляют собой адреса элементов массива. Аналогично, переменные в диапазоне от `p[0]` до `p[a.Length - 1]` представляют собой сами элементы массива. С учетом способа хранения массивов можно рассматривать массив с любым числом измерений как линейный.

Пример:

```
using System;
class Test
{
    static void Main()
```

продолжение ⇗

```
    {
        int[, ,] a = new int[2, 3, 4];
        unsafe
        {
            fixed (int* p = a)
            {
                for (int i = 0; i < a.Length; ++i)    // Рассматриваем как
                                                      // линейный
                        p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j)
            {
                for (int k = 0; k < 4; ++k)
                        Console.Write("[{0},{1},{2}] = {3,2} ", i, j,
                                                        k, a[i, j, k]);
                Console.WriteLine();
            }
    }
}
```

Эта программа выведет следующее:

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

Пример:

```
class Test
{
    unsafe static void Fill(int* p, int count, int value)
    {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main()
    {
        int[] a = new int[100];
        unsafe
        {
                fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

Здесь оператор `fixed` используется для фиксации массива, с тем чтобы его адрес можно было передать в метод, принимающий указатель.

Пример:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}
```

```
class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize)
    {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name)
        {
                PutString("Times New Roman", p, 32);
        }
    }
}
```

Здесь оператор `fixed` используется для фиксации буфера фиксированного размера, находящегося в структуре, с тем чтобы его адрес можно было использовать как указатель.

Значение типа `char*`, получаемое в результате фиксации экземпляра строки, всегда указывает на строку, завершающуюся нулем. Внутри оператора `fixed`, получающего указатель `p` на экземпляр строки `s`, значения указателя, лежащие в диапазоне от `p` до `p + Length - 1`, представляют собой адреса символов в строке, а значение указателя `p + s.Length` всегда указывает на нулевой символ (символ со значением '`\0`').

Изменение объектов управляемого типа с помощью фиксированных указателей может привести к неопределенному поведению. Например, поскольку строки являются неизменяемыми, ответственность за то, что символы, на которые ссылается указатель на фиксированную строку, остаются неизменными, лежит на программисте.

Автоматическое дополнение строк нулевым символом особенно удобно при вызове внешних API, ожидающих строки «в стиле C». Однако заметьте, что в строке допускается наличие нулевых символов. Если они присутствуют, то при обработке строки как последовательности `char*`, завершающейся нулем, она будет усечена.

18.7. Буферы фиксированного размера

Буферы фиксированного размера используются для объявления линейных массивов «в стиле C» в качестве элементов структур и в первую очередь полезны для взаимодействия с неуправляемыми API.

18.7.1. Объявления буферов фиксированного размера

Буфер-фиксированного-размера — это элемент, представляющий собой хранилище для буфера переменных заданного типа, имеющего фиксированный размер. Объявление буфера фиксированного размера создает один или несколько буферов фиксированного размера для элементов заданного типа. Буферы фиксированного размера можно использовать только в объявлениях структур и только в небезопасных контекстах (раздел 18.1).

объявление-элемента-структуры:
    …
    объявление-буфера-фиксированного-размера

объявление-буфера-фиксированного-размера:
    атрибуты$_{opt}$ *модификаторы-буфера-фиксированного-размера*$_{opt}$ **fixed**
        тип-элементов-буфера *описатели-буферов-фиксированного-размера* **;**

модификаторы-буфера-фиксированного-размера:
    модификатор-буфера-фиксированного-размера
    модификатор-буфера-фиксированного-размера
        модификаторы-буфера-фиксированного-размера

модификатор-буфера-фиксированного-размера:
    new
    public
    protected
    internal
    private
    unsafe

тип-элементов-буфера:
    тип

описатели-буферов-фиксированного-размера:
    описатель-буфера-фиксированного-размера
    описатель-буфера-фиксированного-размера **,** *описатели-буферов-фиксированного-размера*

описатель-буфера-фиксированного-размера:
    идентификатор **[** *константное-выражение* **]**

Объявление буфера фиксированного размера может включать в себя набор атрибутов (раздел 17), модификатор `new` (раздел 10.2.2), корректную совокупность четырех модификаторов доступа (раздел 10.2.3) и модификатор `unsafe` (раздел 18.1). Атрибуты и модификаторы применяются ко всем элементам, объявленным в объявлении буфера фиксированного размера. Если один и тот же модификатор встречается в объявлении несколько раз, возникает ошибка компиляции.

Объявление буфера фиксированного размера не может содержать модификатор `static`.

Тип элементов буфера в объявлении буфера фиксированного размера определяет тип элементов в буфере(-ах), создаваемом(-ых) объявлением. Этот тип должен

являться одним из встроенных типов `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double` или `bool`.

За типом элементов буфера следует список описателей буферов фиксированного размера, каждый из которых создает новый элемент структуры. Описатель буфера фиксированного размера состоит из идентификатора, определяющего имя элемента, за которым следует константное выражение, заключенное в лексемы `[` и `]`. Константное выражение определяет число элементов в элементе структуры, создаваемом данным описателем. Тип этого выражения должен быть неявно преобразуем в тип `int`, а его значение должно являться строго положительным целым числом.

Для элементов буфера фиксированного размера гарантируется их последовательное размещение в памяти.

Объявление буфера фиксированного размера, которое объявляет несколько буферов, эквивалентно нескольким объявлениям одного буфера с одинаковыми атрибутами и типами элементов.

Пример:

```
unsafe struct A
{
    public fixed int x[5], y[10], z[100];
}
```

Приведенный код эквивалентен следующему:

```
unsafe struct A
{
    public fixed int x[5];
    public fixed int y[10];
    public fixed int z[100];
}
```

18.7.2. Использование буферов фиксированного размера в выражениях

Поиск элемента (раздел 7.3) для элемента, являющегося буфером фиксированного размера, осуществляется так же, как и для поля.

В выражении на буфер фиксированного размера можно ссылаться, используя *простое-имя* (раздел 7.5.2) или *доступ-к-элементу-объекта* (раздел 7.5.4).

Когда на элемент, являющийся буфером фиксированного размера, ссылаются с использованием простого имени, результат эквивалентен использованию доступа к элементу объекта в форме `this.I`, где `I` — буфер фиксированного размера.

При доступе к элементу объекта в форме `E.I` в случае, когда `E` имеет тип структуры, а результатом поиска элемента `I` в этом структурном типе является элемент, представляющий собой буфер фиксированного размера, `E.I` вычисляется и классифицируется следующим образом:

- Если выражение `E.I` не находится в небезопасном контексте, возникает ошибка компиляции.
- Если `E` классифицируется как значение, возникает ошибка компиляции.

- В противном случае, если `E` является перемещаемой переменной (раздел 18.3), а выражение `E.I` не является *инициализатором-фиксированного-указателя* (раздел 18.6), возникает ошибка компиляции.
- В противном случае `E` ссылается на фиксированную переменную и результатом выражения является указатель на первый элемент буфера фиксированного размера `I` в `E`. Результат имеет тип `S*`, где `S` — тип элементов `I`, и классифицируется как значение.

Доступ к последующим элементам буфера фиксированного размера осуществляется через первый элемент с помощью операций с указателями. В отличие от доступа к массивам, доступ к элементам буфера фиксированного размера является небезопасной операцией, и проверка выхода за пределы буфера не выполняется.

Следующий пример демонстрирует объявление и использование структуры, содержащей буфер фиксированного размера:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize)
    {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

18.7.3. Проверка явного присваивания

Буферы фиксированного размера не подлежат проверке на явное присваивание (5.3), и когда данная проверка выполняется для переменных структурного типа, элементы, являющиеся буферами фиксированного размера, игнорируются.

Когда самая внешняя переменная типа структуры, содержащая буфер фиксированного размера, является статической переменной, переменной экземпляра класса или элементом массива, элементы буфера автоматически инициализируются своими значениями по умолчанию (раздел 5.2). Во всех остальных случаях начальное содержимое буфера фиксированного размера не определено.

18.8. Выделение памяти в стеке

В небезопасном контексте объявление локальной переменной (раздел 8.5.1) может содержать инициализатор выделения памяти в стеке, который выделяет память в стеке вызовов.

> **ЭРИК ЛИППЕРТ**
>
> Эта формулировка является излишне детальной; не существует требования того, чтобы конкретный процессор допускал выделение произвольной памяти в своем *стеке вызовов*. Например, CLR может быть реализована на процессорной архитектуре, использующей *два* стека — один для хранения локальных переменных, а второй для отслеживания адресов возвратов. (Такие архитектуры предотвращают атаки, связанные с модификацией содержимого стека, которым подвержена архитектура x86.) На самом деле в данном случае нам следовало просто сказать, что с каждым вызовом метода ассоциируется неперемещаемая *область локальной памяти* и что `stackalloc` выделяет память из этой области. В типовой реализации такой областью является стек, однако данный факт представляет собой деталь реализации.

инициализатор-локальной-переменной:
    …
    инициализатор-stackalloc

инициализатор-stackalloc:
    stackalloc неуправляемый-тип [выражение]

Неуправляемый-тип обозначает тип элементов, которые будут храниться в созданной области, а *выражение* — их количество. Вместе они определяют требуемый размер области. Поскольку размер выделяемой в стеке памяти не может быть отрицательным, при указании в качестве числа элементов *константного-выражения*, результатом вычисления которого является отрицательное значение, возникает ошибка компиляции.

Инициализатор выделения памяти в стеке в форме `stackalloc T[E]` требует, чтобы `T` являлся неуправляемым типом (раздел 18.2), а `E` — выражением типа `int`. Данная конструкция выделяет в стеке вызовов `E * sizeof(T)` байт и возвращает указатель типа `T*` на только что выделенный блок. Если значение `E` отрицательно, поведение не определено. Если значение `E` равно нулю, выделения памяти не происходит, а значение возвращаемого указателя зависит от реализации. Если для выделения блока заданного размера недостаточно памяти, выбрасывается исключение `System.StackOverflowException`.

Первоначальное содержимое выделенной памяти не определено.

Недопустимо использование инициализаторов выделения памяти в стеке в блоках `catch` и `finally` (раздел 8.10).

Не существует способа явно освободить память, выделенную с помощью `stackalloc`. Все блоки памяти, выделенные в стеке в процессе выполнения элемента-функции, автоматически освобождаются при возврате из этой функции. Такое поведение соответствует функции `alloca` — расширению, часто встречающемуся в реализациях C и C++.

Пример:

```
using System;

class Test
{
    static string IntToString(int value)
    {
        int n = value >= 0 ? value : -value;
        unsafe
        {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do
            {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);

            if (value < 0) *--p = '-';
                return new string(p, 0, (int)(buffer + 16 - p));
            }
    }

    static void Main()
    {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

Здесь инициализатор `stackalloc` используется в методе `IntToString`, чтобы выделить в стеке буфер из `16` символов. Буфер автоматически уничтожается при возврате из метода.

18.9. Динамическое выделение памяти

За исключением операции `stackalloc` в C# не существует встроенных конструкций для управления памятью, в которой не выполняется сборка мусора. Такие возможности обычно предоставляются вспомогательными библиотеками классов или импортируются непосредственно из операционной системы. Например, приведенный далее класс `Memory` демонстрирует, как из C# можно осуществить доступ к функциям взаимодействия с кучей, предоставленным операционной системой:

```
using System;
using System.Runtime.InteropServices;

public unsafe class Memory
{
    // Дескриптор кучи процесса. Этот дескриптор используется
    // во всех вызовах API вида HeapXXX в приведенных далее методах.

    static int ph = GetProcessHeap();
```

```
// Закрытый конструктор экземпляра, предотвращающий
// создание экземпляра этого класса.

private Memory() { }

// Выделяет блок памяти указанного размера.
// Выделенная память автоматически инициализируется нулями.

public static void* Alloc(int size)
{
    void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
    if (result == null) throw new OutOfMemoryException();
    return result;
}

// Копирует count байт из src в dst. Блоки источника
// и назначения могут перекрываться.

public static void Copy(void* src, void* dst, int count)
{
    byte* ps = (byte*)src;
    byte* pd = (byte*)dst;
    if (ps > pd)
    {
            for (; count != 0; count--) *pd++ = *ps++;
    }
    else if (ps < pd)
    {
            for (ps += count, pd += count; count != 0; count--)
                *--pd = *--ps;
    }
}

// Освобождает блок памяти.

public static void Free(void* block)
{
    if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
}

// Осуществляет перераспределение блока памяти.
// Если запрашивается выделение блока большего размера, дополнительная
// область памяти автоматически инициализируется нулями.

public static void* ReAlloc(void* block, int size)
{
    void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
    if (result == null) throw new OutOfMemoryException();
        return result;
}

// Возвращает размер блока памяти.

public static int SizeOf(void* block)
{
    int result = HeapSize(ph, 0, block);
```

продолжение ⇗

```
        if (result == -1) throw new InvalidOperationException();
        return result;
    }

    // Флаги, используемые в API взаимодействия с кучей.

    const int HEAP_ZERO_MEMORY = 0x00000008;

    // Функции из API взаимодействия с кучей.

    [DllImport("kernel32")]
    static extern int GetProcessHeap();

    [DllImport("kernel32")]
    static extern void* HeapAlloc(int hHeap, int flags, int size);

    [DllImport("kernel32")]
    static extern bool HeapFree(int hHeap, int flags, void* block);

    [DllImport("kernel32")]
    static extern void* HeapReAlloc(int hHeap, int flags,
        void* block, int size);

    [DllImport("kernel32")]
    static extern int HeapSize(int hHeap, int flags, void* block);
}
```

Ниже приведен пример, использующий класс `Memory`:

```
class Test
{
    static void Main()
    {
        unsafe
        {
            byte* buffer = (byte*)Memory.Alloc(256);
            try
            {
                for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
                byte[] array = new byte[256];
                fixed (byte* p = array) Memory.Copy(buffer, p, 256);
            }
            finally
            {
                Memory.Free(buffer);
            }
            for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
        }
    }
}
```

В этом примере с помощью `Memory.Alloc` выделяются `256` байт памяти, которые инициализируются последовательностью значений от `0` до `255`. Затем выделяется массив байт, состоящий из `256` элементов, и используется `Memory.Copy` для копирования содержимого блока памяти в этот массив. И наконец, блок памяти очищается с помощью `Memory.Free`, а содержимое массива байт выводится на консоль.

Приложение А
Документирующие комментарии

Язык C# позволяет программистам документировать своей код, используя специальный синтаксис комментариев, содержащих текст в формате XML. В исходных файлах комментарии, имеющие определенную форму, можно использовать для управления инструментом, который из этих комментариев и элементов исходного кода, которым предшествуют комментарии, создает XML. Комментарии, использующие этот синтаксис, называются **документирующими комментариями**. Они должны непосредственно предшествовать определенному пользователем типу (например, классу, делегату или интерфейсу) или элементу (например, полю, событию, свойству или методу). Инструмент, создающий XML, называется **генератором XML**. (Этим генератором может, но не должен быть сам компилятор C#). Генератор дает на выходе файл документации, который используется **средством просмотра документации** — инструментом, предназначенным для создания визуального представления информации о типах и связанной с ними документации.

Спецификация содержит набор тегов для использования в документирующих комментариях. Использование этих тегов не обязательно, при желании можно использовать и другие теги, соответствующие требованиям XML.

А.1. Введение

Комментарии, имеющие специальную форму, можно использовать для получения XML из них и из элементов исходного кода, которым предшествуют комментарии. Это либо однострочные комментарии, начинающиеся с тройного слэша (///), или многострочные комментарии, начинающиеся со слэша и двух звездочек (/**). Они должны непосредственно предшествовать определенному пользователем типу (например, классу, делегату или интерфейсу) или элементу (например, полю, событию, свойству или методу), которые они поясняют. Секции атрибутов (раздел 17.2) рассматриваются как часть объявлений, поэтому документирующие комментарии должны предшествовать атрибутам, применяемым к типу или элементу.

Синтаксис:

однострочный-документирующий-комментарий:
/// *входные-символы*$_{opt}$
многострочный-документирующий-комментарий:
/** *текст-комментария*$_{opt}$ */

Если в *однострочном комментарии*, расположенном рядом с текущим, после символов /// находится пробельный символ, он не включается в результирующий документ XML.

Если в *многострочном комментарии* первый непробельный символ во второй строке является *звездочкой* и одна и та же последовательность необязательных пробельных символов и звездочек повторяется на каждой строке *многострочного комментария*, то символы этой последовательности не включаются в результирующий документ XML. *Пробельные символы* в последовательности могут встречаться как до, так и после *звездочки*.

Пример:

```
/// <summary>Класс <c>Point</c> моделирует точку на плоскости
/// </summary>
///
public class Point
{
   /// <summary>метод <c>draw</c> рисует точку.</summary>
   void draw() {…}
}
```

Текст в документирующих комментариях должен соответствовать правилам XML (http://www.w3.org/TR/REC-xml). Если он не соответствует правилам, выдается предупреждение, а в результирующий файл документации включается комментарий, информирующий о найденной ошибке.

Хотя разработчики могут создавать собственные наборы тегов, в разделе А.2 приведен список рекомендуемых тегов. Некоторые из них имеют специальные значения:

- Тег `<param>` применяется для описания параметров. Если этот тег используется, генератор документации должен проверить существование заданного параметра и то, что все параметры описаны в документирующих комментариях. Если проверка завершается неудачно, генератор документации выдает предупреждение.
- В любом теге можно задать атрибут `cref` для создания ссылки на элемент кода. Генератор документации должен проверить существование этого элемента кода. Если проверка завершается неудачно, генератор документации выдает предупреждение. При поиске имени, описанного в атрибуте `cref`, генератор документации должен учитывать видимость пространств имен в соответствии с операторами `using` в исходном коде. Для обобщенных элементов кода использовать обычный синтаксис (например, `List<T>`) нельзя, поскольку это даст неверный XML. Вместо угловых скобок следует использовать фигурные (например, `List{T}`) или управляющие последовательности XML (например, `List<T>`).
- Тег `<summary>` предназначен для использовании средством просмотра документации для отображения дополнительной информации о типе или элементе.
- Тег `<include>` включает информацию из внешнего файла XML.

Отметим, что файл документации не предоставляет полной информации о типах и элементах (например, он не содержит никакой информации о типах). Для полу-

чения такой информации необходимо использовать файл документации совместно с рефлексией фактического типа или элемента.

А.2. Рекомендуемые теги

Генератор документации воспринимает и обрабатывает любые теги, соответствующие правилам XML. Приведенные в табл. А.1 теги обеспечивают часто используемые возможности документации пользователя. (Можно использовать и другие теги.)

Таблица А.1. Список рекомендуемых тегов

Тег	Раздел	Назначение
`<c>`	А.2.1	Задать шрифт текста, используемый для кода
`<code>`	А.2.2	Задать одну или более строк исходного кода в выводе программы
`<example>`	А.2.3	Задать пример
`<exception>`	А.2.4	Задать исключения, которые может выбрасывать метод
`<include>`	А.2.5	Включить XML из внешнего файла
`<list>`	А.2.6	Создать список или таблицу
`<para>`	А.2.7	Структурировать текст
`<param>`	А.2.8	Описать параметр метода или конструктора
`<paramref>`	А.2.9	Задать слово как имя параметра
`<permission>`	А.2.10	Задать разрешения доступа к элементу
`<remark>`	А.2.11	Описать дополнительную информацию о типе
`<returns>`	А.2.12	Описать возвращаемое значение метода
`<see>`	А.2.13	Задать ссылку
`<seealso>`	А.2.14	Создать ссылку *See Also*
`<summary>`	А.2.15	Описать тип или элемент типа
`<value>`	А.2.16	Описать свойство
`<typeparam>`	А.2.17	Описать параметр обобщенного типа
`<typeparamref>`	А.2.18	Задать слово как имя параметра-типа

А.2.1. <c>

Этот тег обеспечивает возможность задать, что фрагмент текста внутри описания должен отображаться шрифтом, используемым для блока кода. Для строк фактического кода необходимо пользоваться тегом `<code>` (раздел А.2.2).

Синтаксис:

```
<c>текст</c>
```

Пример:

```
/// <summary>Класс <c>Point</c> моделирует точку на плоскости
/// </summary>
public class Point
{
   // ...
}
```

A.2.2. <code>

Этот тег используется для задания вывода одной или более строк исходного кода специальным шрифтом. Для небольших фрагментов кода в тексте используйте `<c>` (раздел А.2.1).

Синтаксис:

```
<code>исходный код вывода программы</code>
```

Пример:

```
/// <summary>Метод изменяет положение точки на
/// заданные смещения x и y.
/// <example>Например:
/// <code>
///    Point p = new Point(3,5);
///    p.Translate(-1,3);
/// </code>
/// в результате <c>p</c> имеет значение (2,8).
/// </example>
/// </summary>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

A.2.3. <example>

Этот тег описывает код примера применения метода или другого элемента библиотеки. Применяется внутри комментария. Обычно при этом также используется тег `<code>` (раздел А.2.2).

Синтаксис:

```
<example>описание</example>
```

Пример:

См. раздел `<code>` (раздел А.2.2).

A.2.4. <exception>

Этот тег позволяет описывать исключения, выбрасываемые методом.

Синтаксис:

```
<exception cref="элемент">описание</exception>
```

где

```
cref="элемент"
```

является именем элемента. Генератор документации проверяет существование данного элемента и преобразует имя в каноническую форму имени, используемую в файле документации.

описание

Описание условий, при которых выбрасывается исключение.

Пример:

```
public class DataBaseOperations
{
   /// <exception cref="MasterFileFormatCorruptException"></exception>
   /// <exception cref="MasterFileLockedOpenException"></exception>
   public static void ReadRecord(int flag) {
      if (flag == 1)
         throw new MasterFileFormatCorruptException();
      else if (flag == 2)
         throw new MasterFileLockedOpenException();
      // …
   }
}
```

A.2.5. <include>

Этот тег включает информацию из документа XML, который является внешним по отношению к данному. Внешний файл должен соответствовать формату XML. Для указания, какой XML из документа включать в данный, к файлу применяется выражение XPath. Тег `<include>` в результате заменяется выбранным XML из внешнего документа.

Синтаксис:

```
<include file="имя файла" path="xpath" />
```

где

`file="`*`имя файла`*`"`

имя внешнего файла XML. Имя файла интерпретируется относительно файла, содержащего включающий тег.

`path="`*`xpath`*`"`

Выражение XPath, выбирающее требуемый XML из внешнего файла XML.

Пример:

Если исходный код содержит объявление:

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { … }
```

а внешний файл docs.xml имеет следующее содержание:

```
<?xml version="1.0"?>
<extradoc>
   <class name="IntList">
      <summary>
         Содержит список целых чисел.
      </summary>
   </class>
   <class name="StringList">
```

продолжение ☞

```
      <summary>
         Содержит список строк.
      </summary>
   </class>
</extradoc>
```

Выдается такая же документация, как если бы исходный код содержал:

```
/// <summary>
/// Содержит список целых чисел.
/// </summary>
public class IntList { … }
```

A.2.6. <list>

Этот тег используется для создания списка или таблицы элементов. Тег может содержать блок `<listheader>` для описания строки заголовков таблицы или списка описаний. (При описании таблицы необходимо указывать в заголовке только вход для *термин*.)

Каждый элемент списка задается с помощью блока `<item>`. При создании списка описаний необходимо задать *термин* и *описание*. Для таблицы, маркированного или нумерованного списка необходимо задавать только *описание*.

Синтаксис:

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>термин</term>
      <description>описание</description>
   </listheader>
   <item>
      <term>термин</term>
      <description>описание</description>
   </item>
   …
   <item>
      <term>термин</term>
      <description>описание</description>
   </item>
</list>
```

где

термин

Определяемый термин, описание которого задается в *описание*.

описание

Либо элемент маркированного или нумерованного списка, либо описание термина.

Пример:

```
public class MyClass
{
   /// <summary>Пример маркированного списка:
   /// <list type="bullet">
   /// <item>
   /// <description>Item 1.</description>
```

```
   /// </item>
   /// <item>
   /// <description>Item 2.</description>
   /// </item>
   /// </list>
   /// </summary>
   public static void Main () {
      // ...
   }
}
```

А.2.7. <para>

Этот тег предназначен для использования внутри других тегов, таких как `<summary>` (раздел А.2.11) и `<returns>` (раздел А.2.12). Он позволяет структурировать текст.

Синтаксис:

```
<para>содержимое</para>
```

где:

содержимое

является текстом абзаца.

Пример:

```
/// <summary>Это точка входа в программу тестирования класса Point.
/// <para>Программа тестирует каждый метод и операцию
/// и предназначена для запуска после всякого нетривиального изменения
/// класса Point.</para></summary>
public static void Main() {
   // ...
}
```

А.2.8. <param>

Этот тег используется для описания параметра метода, конструктора или индексатора.

Синтаксис:

```
<param name="имя">описание</param>
```

где

имя

Имя параметра.

описание

Описание параметра.

Пример:

```
/// <summary>Метод изменяет положение точки,
/// используя заданные координаты </summary>
/// <param name="xor">новая координата x.</param>
/// <param name="yor"> новая координата y.</param>
```

продолжение ⇗

```
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

А.2.9. <paramref>

Этот тег используется для указания, что заданное слово является параметром. Файл документации может быть обработан для форматирования этого параметра каким-либо особым образом.

Синтаксис:

```
<paramref name="имя"/>
```

где

имя

Имя параметра.

Пример:

```
/// <summary>Конструктор инициализирует новую точку значениями
/// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor"> новая координата x.</param>
/// <param name="yor"> новая координата y.</param>
public Point(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

А.2.10. <permission>

Этот тег задает разрешения доступа к документируемому элементу.

Синтаксис:

```
<permission cref="элемент">описание</permission>
```

где

`cref="элемент"`

Имя элемента. Генератор документации проверяет существование данного элемента и преобразует имя в каноническую форму имени, используемую в файле документации.

описание

Описание доступа к элементу.

Пример:

```
/// <permission cref="System.Security.PermissionSet">Доступ
/// разрешен для всех.</permission>
public static void Test() {
   // ...
}
```

А.2.11. <remark>

Этот тег используется для задания дополнительной информации о типе. (Используйте <summary> (раздел А.2.15) для описания самого типа и его элементов.)

Синтаксис:

```
<remark>описание</remark>
```

где

описание

Текст описания.

Пример:

```
/// <summary>Класс <c>Point</c> моделирует точку
/// на плоскости.</summary>
/// <remark>Использует полярные координаты</remark>
public class Point
{
   // ...
}
```

А.2.12. <returns>

Этот тег используется для описания возвращаемого значения метода.

Синтаксис:

```
<returns>описание</returns>
```

где

описание

Описание возвращаемого значения.

Пример:

```
/// <summary>Выводит положение точки в виде строки.</summary>
/// <returns>Строковое представление положения точки в форме (x,y),
/// без ведущих, завершающих и промежуточных пробелов.</returns>
public override string ToString() {
   return "(" + X + "," + Y + ")";
}
```

А.2.13. <see>

Этот тег позволяет задать ссылку внутри текста. Используйте `<seealso>` (раздел А.2.14) для описания текста, который должен отображаться в секции *See Also*.

Синтаксис:

```
<see cref="элемент"/>
```

где

`cref="элемент"`

Имя элемента. Генератор документации проверяет существование данного элемента и преобразует имя в каноническую форму имени, используемую в файле документации.

Пример:

```
/// <summary>Метод изменяет положение точки,
/// используя заданные координаты </summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}
/// <summary>Метод изменяет положение точки на
/// заданные смещения x и y.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

A.2.14. <seealso>

Этот тег позволяет создать ссылку для секции *See Also*. Используйте `<see>` (раздел A.2.13) для задания ссылки внутри текста.

Синтаксис:

```
<seealso cref="элемент"/>
```

где

cref="элемент"

Имя элемента. Генератор документации проверяет существование данного элемента и преобразует имя в каноническую форму имени, используемую в файле документации.

Пример:

```
/// <summary>Метод определяет, имеют ли две точки
/// одно и то же положение.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
   // ...
}
```

A.2.15. <summary>

Этот тег можно использовать для краткого описания типа или элемента типа. Используйте `<remark>` (раздел A.2.11) для описания самого типа.

Синтаксис:

```
<summary>описание</summary>
```

где

описание

Описание типа или элемента.

Пример:

```
/// <summary>Конструктор инициализирует точку в (0,0).</summary>
public Point() : this(0,0) {
}
```

А.2.16. <value>

Этот тег позволяет описать свойство.

Синтаксис:

```
<value>описание свойства</value>
```

где

описание свойства

Описание свойства.

Пример:

```
/// <value>Свойство <c>X</c> представляет координату x точки.</value>
public int X
{
   get { return x; }
   set { x = value; }
}
```

А.2.17. <typeparam>

Этот тег используется для описания параметра обобщенного типа для класса, структуры, интерфейса, делегата или метода.

Синтаксис:

```
<typeparam name="имя">описание</typeparam>
```

где

name

Имя параметра-типа.

описание

Описание параметра-типа.

Пример:

```
/// <summary>Класс обобщенного списка.</summary>
/// <typeparam name="T">Тип элементов списка.</typeparam>
public class MyList<T> {
   ...
}
```

А.2.18. <typeparamref>

Этот тег используется для указания, что заданное слово является параметром-типом. Файл документации может быть обработан для форматирования этого параметра каким-либо особым образом.

Синтаксис:

```
<typeparamref name="имя"/>
```

где

имя

Имя параметра-типа.

Пример:

```
/// <summary>Метод выполняет выборку данных и возвращает
/// список из <typeparamref name="T"> "/>"> .</summary>
/// <param name="string">запрос для выполнения</param>
public List<T> FetchData<T>(string query) {
   ...
}
```

А.3. Обработка файла документации

Генератор документации создает для каждого элемента исходного кода ID-строку, связанную с документирующим комментарием. Эта ID-строка однозначно определяет исходный элемент. Средство просмотра документации может использовать ID-строку для идентификации соответствующих метаданных или элемента рефлексии, для которых применяется документация.

Файл документации не является иерархическим представлением исходного кода, напротив, это плоский список со сгенерированной ID-строкой для каждого элемента.

А.3.1. Формат ID-строки

При создании ID-строки генератор документации руководствуется следующими правилами:

- Пробелы в строке отсутствуют.
- Первая часть строки определяет тип документируемого элемента с помощью одиночного символа, за которым следует двоеточие. Типы элементов перечислены в табл. А.2.

Таблица А.2. Типы документируемых элементов

Символ	**Описание**
E	Событие
F	Поле
M	Метод (включая конструкторы, деструкторы и операции)
N	Пространство имен
P	Свойство
T	Тип (например, класс, делегат, перечисление, интерфейс или структура)
!	Строка ошибки. Остаток строки содержит информацию об ошибке. Например, генератор документации создает информацию об ошибках для неразрешенных ссылок.

- Вторая часть строки задает полное имя элемента, начиная с корня пространства имен. Имя элемента, содержащий его тип и пространство имен разделены точками. Если имя самого элемента содержит точки, они заменяются символами с кодом `#` (U+0023). (Предполагается, что таких символов нет ни в одном элементе.)
- Для методов и свойств с аргументами далее следует список аргументов, заключенный в круглые скобки. Если аргументы отсутствуют, скобки не указываются. Аргументы разделены запятыми. Кодировка каждого аргумента соответствует сигнатуре CLI следующим образом:
  - Аргумент представлен документирующим именем, основанным на полном имени аргумента, преобразованном следующим образом:
    - Имя аргумента, представляющего обобщенный тип, дополняется символом «`'`», за которым следует количество параметров-типов.
    - Имя типа аргумента, содержащего модификатор `ref` или `out`, дополняется символом `@`. Аргументы, передаваемые по значению или через `params`, не имеют специального обозначения.
    - Имя аргумента, представляющего собой массив, представляется как `[` *нижняя граница* `:` *размер* `,` ... `,` *нижняя граница* `:` *размер* `]`, где количество запятых соответствует рангу минус 1, а *нижняя граница* и *размер*, если они известны, представлены в десятичной форме. Если *нижняя граница* или *размер* для какой-либо размерности не указаны, двоеточие также опускается. Ступенчатые массивы представляются одним «`[]`» для каждого уровня.
    - Имя типа аргумента типа указателя, если это не указатель на `void`, дополняется `*`. Указатель на представлен с использованием имени типа `System.Void`.
    - Аргументы, соответствующие параметрам обобщенных типов, определенным в типах, кодируются с помощью символа обратного апострофа «`` ` ``», за которым следует порядковый номер параметра-типа (нумерация начинается с нуля).
    - Аргументы, соответствующие параметрам обобщенных типов, определенным в методах, кодируются с помощью двух обратных апострофов «``` `` ```» вместо одного, который используется для типов.

A.3.2. Примеры ID-строк

В следующих примерах представлены фрагменты кода и соответствующие им ID-строки для каждого исходного элемента, способного иметь документирующие комментарии

- Типы представлены с помощью полного имени, дополненного информацией об обобщениях:

```
enum Color { Red, Blue, Green }
namespace Acme
{
   interface IProcess {...}
   struct ValueType {...}
   class Widget: IProcess
   {
      public class NestedClass {...}
      public interface IMenuItem {...}
      public delegate void Del(int i);
      public enum Direction { North, South, East, West }
   }
   class MyList<T>
   {
      class Helper<U,V> {...}
   }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"
```

- Поля представлены с помощью полного имени:

```
namespace Acme
{
   struct ValueType
   {
      private int total;
   }
   class Widget: IProcess
   {
      public class NestedClass
      {
         private int value;
      }
      private string message;
      private static Color defaultColor;
      private const double PI = 3.14159;
      protected readonly double monthlyAverage;
      private long[] array1;
      private Widget[,] array2;
      private unsafe int *pCount;
      private unsafe float **ppValues;
   }
}
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
```

```
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Конструкторы:

```
namespace Acme
{
   class Widget: IProcess
   {
      static Widget() {...}
      public Widget() {...}
      public Widget(string s) {...}
   }
}
"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Деструкторы:

```
namespace Acme
{
   class Widget: IProcess
   {
      ~Widget() {...}
   }
}
"M:Acme.Widget.Finalize"
```

- Методы:

```
namespace Acme
{
   struct ValueType
   {
      public void M(int i) {...}
   }
   class Widget: IProcess
   {
      public class NestedClass
      {
         public void M(int i) {...}
      }
      public static void M0() {...}
      public void M1(char c, out float f, ref ValueType v) {...}
      public void M2(short[] x1, int[,] x2, long[][] x3) {...}
      public void M3(long[][] x3, Widget[][,,] x4) {...}
      public unsafe void M4(char *pc, Color **pf) {...}
      public unsafe void M5(void *pv, double *[][,] pd) {...}
      public void M6(int i, params object[] args) {...}
   }
   class MyList<T>
   {
      public void Test(T t) { }
   }
```

продолжение ➪

```
   class UseList
   {
      public void Process(MyList<int> list) { }
      public MyList<T> GetValues<T>(T inputValue) { return null; }
   }
}
"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues``(``0)"
```

- Свойства и индексаторы:

```
namespace Acme
{
   class Widget: IProcess
   {
      public int Width { get {...} set {...} }
      public int this[int i] { get {...} set {...} }
      public int this[string s, int i] { get {...} set {...} }
   }
}
"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- События:

```
namespace Acme
{
   class Widget: IProcess
   {
      public event Del AnEvent;
   }
}
"E:Acme.Widget.AnEvent"
```

- Унарные операции:

```
namespace Acme
{
   class Widget: IProcess
   {
      public static Widget operator+(Widget x) {...}
   }
}
"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

Полный набор имен унарных функций-операций: `op_UnaryPlus`, `op_UnaryNegation`, `op_LogicalNot`, `op_OnesComplement`, `op_Increment`, `op_Decrement`, `op_True`, `op_False`.

- Бинарные операции:

```
namespace Acme
{
   class Widget: IProcess
   {
      public static Widget operator+(Widget x1, Widget x2) {...}
   }
}
"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

Полный набор имен бинарных функций-операций: `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_Modulus`, `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`, `op_LeftShift`, `op_RightShift`, `op_Equality`, `op_Inequality`, `op_LessThan`, `op_LessThanOrEqual`, `op_GreaterThan`, `op_GreaterThanOrEqual`.

- Операции приведения заканчиваются символом `~`, за которым следует тип возвращаемого значения:

```
namespace Acme
{
   class Widget: IProcess
   {
      public static explicit operator int(Widget x) {...}
      public static implicit operator long(Widget x) {...}
   }
}
"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

А.4. Пример

А.4.1. Исходный код на C#

В следующем примере приведен исходный код класса `Point`:

```
namespace Graphics
{
   /// <summary>Класс <c>Point</c> моделирует точку на плоскости.
   /// </summary>
   public class Point
   {
      /// <summary>Переменная экземпляра <c>x</c> задает координату
      /// x точки.</summary>
      private int x;
      /// <summary>Переменная экземпляра <c>y</c> задает координату
      /// y точки.</summary>
      private int y;
      /// <value>Свойство <c>X</c> задает координату x точки.</value>
      public int X
      {
         get { return x; }
         set { x = value; }
      }
```

продолжение ☞

```
/// <value>Свойство <c>Y</c> задает
/// координату y точки.</value>
public int Y
{
   get { return y; }
   set { y = value; }
}
/// <summary>Конструктор инициализирует точку в (0,0).
/// </summary>
public Point() : this(0, 0) { }
/// <summary> Конструктор инициализирует точку
/// в (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param><c>xor</c> новая координата x точки.</param>
/// <param><c>yor</c> новая координата y точки.</param>
public Point(int xor, int yor)
{
   X = xor;
   Y = yor;
}
/// <summary>Метод изменяет положение точки
/// на заданные координаты.
/// <param><c>xor</c> новая координата x точки.</param>
/// <param><c>yor</c> новая координата y точки.</param>
/// <see cref="Translate"/>
public void Move(int xor, int yor)
{
   X = xor;
   Y = yor;
}
/// <summary>Метод изменяет положение точки
/// на заданные смещения x и y.
/// <example>Например:
/// <code>
/// Point p = new Point(3,5);
/// p.Translate(-1,3);
/// </code>
/// в результате <c>p</c> имеет значение (2,8).
/// </example>
/// </summary>
/// <param><c>xor</c> относительное смещение по x.</param>
/// <param><c>yor</c> относительное смещение по y.</param>
/// <see cref="Move"/>
public void Translate(int xor, int yor)
{
   X += xor;
   Y += yor;
}
/// <summary>Метод определяет, имеют ли две точки
/// одно и то же положение.</summary>
/// <param><c>o</c> сравнивается с текущим объектом.
/// </param>
/// <returns>True если точки имеют одинаковое положение
/// и в точности одинаковый тип, иначе false.</returns>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o)
```

```
{
   if (o == null)
   {
      return false;
   }
   if (this == o)
   {
      return true;
   }
   if (GetType() == o.GetType())
   {
      Point p = (Point)o;
      return (X == p.X) && (Y == p.Y);
   }
   return false;
}
/// <summary>Выводит положение точки в виде строки.</summary>
/// <returns>Строковое представление положения
/// точки в форме (x,y), без ведущих, завершающих
/// и промежуточных пробелов.</returns>
public override string ToString()
{
   return "(" + X + "," + Y + ")";
}
/// <summary> Операция определяет, имеют ли две точки
/// одно и то же положение </summary>
/// <param><c>p1</c> первая точка для сравнения.</param>
/// <param><c>p2</c> вторая точка для сравнения.</param>
/// <returns>True если точки имеют одинаковое положение
/// и в точности одинаковый тип, иначе false.</returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator!="/>
public static bool operator ==(Point p1, Point p2)
{
   if ((object)p1 == null || (object)p2 == null)
   {
      return false;
   }
   if (p1.GetType() == p2.GetType())
   {
      return (p1.X == p2.X) && (p1.Y == p2.Y);
   }
return false;
}
/// <summary> Операция определяет, имеют ли две точки
/// одно и то же положение </summary>
/// <param><c>p1</c> первая точка для сравнения.</param>
/// <param><c>p2</c> вторая точка для сравнения.</param>
/// <returns>True если точки имеют не одинаковое положение
/// и не одинаковый тип, иначе false.</returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator=="/>
public static bool operator !=(Point p1, Point p2)
{
   return !(p1 == p2);
}
```

продолжение ⇨

```
      /// <summary>Это точка входа в программу тестирования класса Point.
      /// <para>Программа тестирует каждый метод и операцию
      /// и предназначена для запуска после всякого нетривиального
      /// изменения класса Point.</para></summary>
      public static void Main()
      {
         // Здесь код тестирования класса
      }
   }
}
```

А.4.2. Результирующий XML

Далее представлен результат работы некоторого генератора документации для исходного кода класса `Point`, приведенного выше:

```
<?xml version="1.0"?>
<doc>
  <assembly>
     <name>Point</name>
  </assembly>
  <members>
   <member name="T:Graphics.Point">
      <summary>
         Class <c>Point</c> моделирует точку на плоскости.
      </summary>
   </member>
   <member name="F:Graphics.Point.x">
      <summary>
         Переменная экземпляра <c>x</c> задает координату
         x точки.
      </summary>
   </member>
   <member name="F:Graphics.Point.y">
      <summary>
         Переменная экземпляра <c>y</c> задает координату
         y точки.
      </summary>
   </member>
   <member name="M:Graphics.Point.#ctor">
      <summary>
         Конструктор инициализирует точку в (0,0).
      </summary>
   </member>
   <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
      <summary>
         Конструктор инициализирует точку в
         (<paramref name="xor"/>,<paramref name="yor"/>).
      </summary>
      <param>
         <c>xor</c> новая координата x точки.
      </param>
      <param>
         <c>yor</c> новая координата y точки.
      </param>
```

```
   </member>
   <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
      <summary>
         Метод изменяет положение точки
         на заданные координаты.
      </summary>
      <param>
         <c>xor</c> новая координата x точки.
      </param>
      <param>
         <c>yor</c> новая координата y точки.
      </param>
      <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
   </member>
   <member
      name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
      <summary>
         Метод изменяет положение точки
         на заданное смещение x и y
         <example>
            Например:
            <code>
               Point p = new Point(3,5);
               p.Translate(-1,3);
            </code>
            в результате <c>p</c> имеет значение (2,8)/
         </example>
      </summary>
      <param>
         <c>xor</c> относительное смещение по x.
      </param>
      <param>
         <c>yor</c> относительное смещение по y.
      </param>
      <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
   </member>
   <member name="M:Graphics.Point.Equals(System.Object)">
      <summary>
         Метод определяет, имеют ли две точки
         одно и то же положение.
      </summary>
      <param>
         <c>o</c> сравнивается с текущим объектом.
      </param>
      <returns>
         True, если точки имеют одинаковое положение
         и в точности одинаковый тип, иначе false.
      </returns>
      <seealso
cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
      <seealso
cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
   </member>
   <member name="M:Graphics.Point.ToString">
      <summary> Выводит положение точки в виде строки.</summary>
      <returns>
```

продолжение ⇗

```
         Строковое представление положения точки в форме (x,y),
         без ведущих, завершающих и промежуточных пробелов.
      </returns>
   </member>
   <member
name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
      <summary>
         Операция определяет, имеют ли две точки
         одно и то же положение.
      </summary>
      <param>
         <c>p1</c> первая точка для сравнения.
      </param>
      <param>
         <c>p2</c> вторая точка для сравнения.
      </param>
      <returns>
         True, если точки имеют одинаковое положение
         и в точности одинаковый тип, иначе false.
      </returns>
      <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
      <seealso
cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
   </member>
   <member
name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
      <summary>
         Операция определяет, имеют ли две точки
         одно и то же положение.
      </summary>
      <param>
         <c>p1</c> первая точка для сравнения.
      </param>
      <param>
         <c>p2</c> вторая точка для сравнения.
      </param>
      <returns>
         True, если точки имеют не одинаковое положение
         и не одинаковый тип, иначе false.
      </returns>
      <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
      <seealso
cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
   </member>
   <member name="M:Graphics.Point.Main">
      <summary>
         Это точка входа в программу тестирования класса Point.
         <para>
            Программа тестирует каждый метод и операцию
            и предназначена для запуска после всякого нетривиального
            изменения класса Point.
         </para>
      </summary>
   </member>
   <member name="P:Graphics.Point.X">
      <value>
```

```
         Свойство <c>X</c> задает координату
         x точки.
      </value>
   </member>
   <member name="P:Graphics.Point.Y">
      <value>
         Свойство <c>Y</c> задает координату
         y точки.
      </value>
   </member>
  </members>
</doc>
```

Приложение Б

Грамматика

Это приложение содержит свод всех правил лексической и синтаксической грамматики из основной части книги, а также грамматические расширения для небезопасного кода. Грамматические правила приведены в том же порядке, в котором они встречаются в основной части книги.

Б.1. Лексическая грамматика

исходный-текст:
    секция-исходного-текста $_{opt}$

секция-исходного-текста:
    часть-секции-исходного-текста
    секция-исходного-текста   часть-секции-исходного-текста

часть-секции-исходного-текста:
    входные-элементы$_{opt}$ *новая-строка*
    директива-препроцессора

входные-элементы:
    входной-элемент
    входные-элементы   входной-элемент

входной-элемент:
    пробельный-символ
    комментарий
    лексема

Б.1.1. Символы конца строки

новая-строка:
    Символ возврата каретки (U+000D)
    Символ перевода строки (U+000A)
    Символ возврата каретки (U+000D), за которым следует символ перевода строки (U+000A)
    Символ новой строки (U+0085)
    Символ разделения строк (U+2028)
    Символ разделения абзацев (U+2029)

Б.1.2. Комментарии

комментарий:
    однострочный-комментарий
    многострочный-комментарий

однострочный-комментарий:
    //   *входные-символы*$_{opt}$

входные-символы:
    входной-символ
    входные-символы   *входной-символ*

входной-символ:
    Любой символ Unicode, кроме *символа-новой-строки*

символ-новой-строки:
    Символ возврата каретки (U+000D)
    Символ перевода строки (U+000A)
    Символ новой строки (U+0085)
    Символ разделения строк (U+2028)
    Символ разделения абзацев (U+2029)

многострочный-комментарий:
    /*   *текст-многострочного-комментария*$_{opt}$ *звездочки* **/**

текст-многострочного-комментария:
    секция-многострочного-комментария
    текст-многострочного-комментария   *секция-многострочного-комментария*

секция-многострочного-комментария:
    /   *звездочки*$_{opt}$ *не-слэш-или-звездочка*

звездочки:
    *****
    звездочки   *****

не-слэш-или-звездочка:
    Любой символ Unicode, кроме **/** и *****

Б.1.3. Пробельные символы

пробельный-символ:
    Любой символ Unicode класса Zs
    Символ горизонтальной табуляции (U+0009)
    Символ вертикальной табуляции (U+000B)
    Символ новой страницы (U+000C)

Б.1.4. Лексемы

лексема:
    идентификатор
    ключевое-слово
    целочисленный-литерал
    вещественный-литерал
    символьный-литерал
    строковый-литерал
    знак-операции-или-пунктуации

Б.1.5. Управляющие последовательности Unicode

управляющая-последовательность-unicode:
\u *шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра*
\U *шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра шестнадцатеричная-цифра*

Б.1.6. Идентификаторы

идентификатор:
доступный-идентификатор
@ *идентификатор-или-ключевое-слово*

доступный-идентификатор:
идентификатор-или-ключевое-слово, не являющийся *ключевым-словом*

идентификатор-или-ключевое-слово:
символ-начала-идентификатора *символы-идентификатора*$_{opt}$

символ-начала-идентификатора:
буквенный-символ
_ (символ подчеркивания U+005F)

символы-идентификатора:
символ-идентификатора
символы-идентификатора *символ-идентификатора*

символ-идентификатора:
буквенный-символ
десятичная-цифра
символ-соединения
символ-комбинирования
символ-управления-форматом

буквенный-символ:
Символ Unicode классов Lu, Ll, Lt, Lm, Lo и Nl
управляющая-последовательность-unicode, представляющая символ классов Lu, Ll, Lt, Lm, Lo и Nl

символ-комбинирования:
Символ Unicode классов Mn или Mc
управляющая-последовательность-unicode, представляющая символ классов Mn или Mc

десятичная-цифра:
символ Unicode класса Nd
управляющая-последовательность-unicode, представляющая символ класса Nd

символ-соединения:
символ Unicode класса Pc
управляющая-последовательность-unicode, представляющая символ класса Pc

символ-управления-форматом:
Символ Unicode класса Cf
управляющая-последовательность-unicode, представляющая символ класса Cf

Б.1.7. Ключевые слова

ключевое-слово: одно из

abstract	**as**	**base**	**bool**	**break**
byte	**case**	**catch**	**char**	**checked**
class	**const**	**continue**	**decimal**	**default**
delegate	**do**	**double**	**else**	**enum**
event	**explicit**	**extern**	**false**	**finally**
fixed	**float**	**for**	**foreach**	**goto**
if	**implicit**	**in**	**int**	**interface**
internal	**is**	**lock**	**long**	**namespace**
new	**null**	**object**	**operator**	**out**
override	**params**	**private**	**protected**	**public**
readonly	**ref**	**return**	**sbyte**	**sealed**
short	**sizeof**	**stackalloc**	**static**	**string**
struct	**switch**	**this**	**throw**	**true**
try	**typeof**	**uint**	**ulong**	**unchecked**
unsafe	**ushort**	**using**	**virtual**	**void**
volatile	**while**			

Б.1.8. Литералы

литерал:
булевский-литерал
целочисленный-литерал
вещественный-литерал
символьный-литерал
строковый-литерал
нулевой-литерал

булевский-литерал:
true
false

целочисленный-литерал:
десятичный-целочисленный-литерал
шестнадцатеричный-целочисленный-литерал

десятичный-целочисленный-литерал:
десятичные-цифры *суффикс-целого-типа*$_{opt}$

десятичные-цифры:
десятичная-цифра
десятичные-цифры *десятичная-цифра*

десятичные-цифры: одно из
0 1 2 3 4 5 6 7 8 9

суффикс-целого-типа: одно из
U u L l UL Ul uL ul LU Lu lU lu

шестнадцатеричный-целочисленный-литерал:
0x *шестнадцатеричные-цифры* *суффикс-целого-типа*$_{opt}$
0X *шестнадцатеричные-цифры* *суффикс-целого-типа*$_{opt}$

шестнадцатеричные-цифры:
шестнадцатеричная-цифра
шестнадцатеричные-цифры *шестнадцатеричная-цифра*

шестнадцатеричная-цифра: одно из
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

вещественный-литерал:
десятичные-цифры **.** *десятичные-цифры* *экспоненциальная-часть*$_{opt}$
суффикс-вещественного-типа$_{opt}$
. *десятичные цифры* *экспоненциальная-часть*$_{opt}$
суффикс-вещественного-типа$_{opt}$
десятичные-цифры *экспоненциальная-часть* *суффикс-вещественного-типа*$_{opt}$
десятичные-цифры *суффикс-вещественного-типа*

экспоненциальная-часть:
e *знак* $_{opt}$ *десятичные-цифры*
E *знак*$_{opt}$ *десятичные- цифры*

знак: одно из
+ -

суффикс-вещественног-типа: одно из
F f D d M m

символьный-литерал:
' *символ* **'**

символ:
единственный-символ
простая-управляющая-последовательность
шестнадцатеричная-управляющая-последовательность
управляющая-последовательность-unicode

единственный-символ:
Любой символ, кроме ' (U+0027), \ (U+005C) и *символа-новой-строки*

простая-управляющая-последовательность: одно из
\' \" \\ \0 \a \b \f \n \r \t \v

шестнадцатеричная-управляющая-последовательность:
\x *шестнадцатеричная-цифра* *шестнадцатеричная-цифра*$_{opt}$
шестнадцатеричная-цифра$_{opt}$ *шестнадцатеричная-цифра*$_{opt}$

строковый-литерал:
обычный-строковый-литерал
буквальный-строковый-литерал

обычный-строковый-литерал:
" *символы-обычного-строкового-литерала*$_{opt}$ **"**

символы-обычного-строкового-литерала:
символ-обычного-строкового-литерала
символы-обычного-строкового-литерала *символ-обычного-строкового-литерала*

символ-обычного-строкового-литерала:
    одиночный-символ-обычного-строкового-литерала
    простая-управляющая-последовательность
    шестнадцатеричная-управляющая-последовательность
    управляющая-последовательность-unicode

одиночный-символ-обычного-строкового-литерала:
    Любой символ, за исключением " (U+0022), \ (U+005C) и *символа-новой-строки*

буквальный-строковый-литерал:
    @" *символы-буквального-строкового-литерала*$_{opt}$ **"**

символы-буквального-строкового-литерала:
    символ-буквального-строкового-литерала
    символы-буквального-строкового-литерала
    символ-буквального-строкового-литерала

символ-буквального-строкового-литерала:
    одиночный-символ-буквального-строкового-литерала
    управляющая-последовательность-кавычки

одиночный-символ-буквального-строкового-литерала:
    Любой символ за исключением "

управляющая-последовательность-кавычки:
    ""

нулевой-литерал:
    null

Б.1.9. Знаки операций и пунктуации

знак-операции-или-пунктуации: одно из

```
{   }   [   ]   (   )   .   ,   :   ;
+   -   *   /   %   &   |   ^   !   ~
=   <   >   ?   ??  ::  ++  --  &&  ||
->  ==  !=  <=  >=  +=  -=  *=  /=  %=
&=  |=  ^=  <<  <<= =>
```

сдвиг-вправо:
    >|>

присваивание-со-сдвигом-вправо:
    >|>=

Б.1.10. Директивы препроцессора

директива-препроцессора:
    директива-объявления
    условная-директива
    директива-line
    директива-диагностики
    директива-region
    директива-pragma

символ-условной-компиляции:
    Любой *идентификатор-или-ключевое-слово* за исключением **true** и **false**
выражение-препроцессора:
    пробельный-символ$_{opt}$ *выражение-or* *пробельный-символ*$_{opt}$

выражение-or:
    выражение-and
    выражение-or *пробельный-символ*$_{opt}$ **||** *пробельный-символ*$_{opt}$ *выражение-and*

выражение-and:
    выражение-равенства
    выражение-and *пробельный-символ*$_{opt}$ **&&** *пробельный-символ*$_{opt}$ *выражение-равенства*

выражение-равенства:
    унарное-выражение-препроцессора
    выражение-равенства *пробельный-символ*$_{opt}$ **==** *пробельный-символ*$_{opt}$ *унарное-выражение-препроцессора*
    выражение-равенства *пробельный-символ*$_{opt}$ **!=** *пробельный-символ*$_{opt}$
унарное-
    выражение-препроцессора

унарное-выражение-препроцессора:
    первичное-выражение-препроцессора
    ! *пробельный-символ*$_{opt}$ *унарное-выражение-препроцессора*

первичное-выражение-препроцессора:
    true
    false
    символ-условной-компиляции
    (*пробельный-символ*$_{opt}$ *выражение-препроцессора* *пробельный-символ*$_{opt}$ **)**

директива-объявления:
    пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **define** *пробельный-символ* *условный-символ* *директива-новой-строки*
    пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **undef** *пробельный-символ* *условный-символ* *директива-новой-строки*

директива-новой-строки:
    пробельный-символ$_{opt}$ *однострочный-комментарий*$_{opt}$ *новая-строка*

условная-директива:
    секция-if-директивы *секции--elif-директивы*$_{opt}$ *секция-else-директивы*$_{opt}$ *секция-endif-директивы*

секция-if-директивы:
    пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **if** *пробельный-символ* *выражение-препроцессора* *директива-новой-строки*
    условная-секция$_{opt}$

секции-elif-директивы:
    секция-elif-директивы
    секции-elif-директивы *секция-elif-директивы*

секция-elif-директивы :
    пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **elif** *пробельный-символ* *выражение-препроцессора* *директива-новой-строки*
    условная-секция$_{opt}$

секция-else-директивы:
пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **else**
директива-новой-строки *условная-секция*$_{opt}$

секция-endif:
пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **endif**
директива-новой-строки

условная-секция:
секция-исходного-текста
пропущенная-секция

пропущенная-секция:
часть-пропущенной-секции
пропущенная-секция *часть-пропущенной-секции*

часть-пропущенной-секции:
пропущенные-символы$_{opt}$ *новая-строка*
директива-препроцессора

пропущенные-символы:
пробельный-символ$_{opt}$ *нечисловой-символ* *входные-символы*$_{opt}$

нечисловой-символ:
Любой *входной-символ*, кроме **#**

директива-диагностики:
пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **error**
сообщение-препроцессора
пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **warning**
сообщение-препроцессора

сообщение-препроцессора:
новая-строка
пробельный-символ *символы-исходного-текста*$_{opt}$ *новая-строка*

директива-region:
начало-фрагмента *условная-секция*$_{opt}$ *конец-фрагмента*

начало-фрагмента:
пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **region**
сообщение-препроцессора

конец-фрагмента:
пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **endregion**
сообщение-препроцессора

директива-line:
пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **line** *пробельный-символ*
индикатор-строки *новая-строка*

индикатор-строки:
десятичные-цифры *пробельный-символ* *имя-файла*
десятичные-цифры
default
hidden

имя-файла:
  " *символы-имени-файла* "

символы-имени-файла:
  символ-имени-файла
  символы-имени-файла *символ-имени-файла*

символ-имени-файла:
  Любой *входной-символ* за исключением "

директива-pragma:
  пробельный-символ$_{opt}$ **#** *пробельный-символ*$_{opt}$ **pragma** *пробельный-символ*
    тело-pragma *новая-строка*

тело-pragma:
  тело-pragma-warning

тело-pragma-warning:
  warning *пробельный-символ* *действие-warning*
  warning *пробельный-символ* *действие-warning* *пробельный-символ*
    список-warning

действие-warning:
  disable
  restore

список-warning:
  десятичные-цифры
  список-warning *пробельный-символ*$_{opt}$ **,** *пробельный-символ*$_{opt}$
    десятичные-цифры

Б.2. Синтаксическая грамматика

Б.2.1. Основные понятия

имя-пространства-имен:
  имя-пространства-имен-или-типа

имя-типа:
  имя-пространства-имен-или-типа

имя-пространства-имен-или-типа:
  идентификатор *список-аргументов-типа*$_{opt}$
  имя-пространства-имен-или-типа **.** *идентификатор* *список-аргументов-типа*$_{opt}$
  уточненный-псевдоним-элемента

Б.2.2. Типы

тип:
  тип-значение
  ссылочный-тип
  параметр-тип

тип-значение:
  структура
  перечисление

структура:
  имя-типа
  простой-тип
  обнуляемый-тип

простой-тип:
  арифметический-тип
  bool

арифметический-тип:
  целочисленный-тип
  вещественный-тип
  decimal

целочисленный-тип:
  sbyte
  byte
  short
  ushort
  int
  uint
  long
  ulong
  char

вещественный-тип:
  float
  double

обнуляемый-тип:
  необнуляемый-тип-значение **?**

необнуляемый-тип-значение:
  тип

перечисление:
  имя-типа

ссылочный-тип:
  класс
  интерфейс
  массив
  делегат

класс:
  имя-типа
  object
  dynamic
  string

интерфейс:
  имя-типа

массив:
  не-массив *спецификаторы-размерности*

не-массив:
  тип

спецификаторы-размерности:
  спецификатор-размерности
  спецификаторы-размерности *спецификатор-размерности*

спецификатор-размерности:
  [*разделители-размерности*$_{opt}$ **]**

разделители-размерности:
  ,
  разделители-размерности **,**

делегат:
  имя-типа

список-аргументов-типов:
  < *аргументы-типы* **>**

аргументы-типы:
  аргумент-тип
  аргументы-типы **,** *аргумент-тип*

аргумент-тип:
  тип

параметр-тип:
  идентификатор

Б.2.3. Переменные

ссылка-на-переменную:
  выражение

Б.2.4. Выражения

список-аргументов:
  аргумент
  список-аргументов **,** *аргумент*

аргумент:
  имя-аргумента$_{opt}$ *значение-аргумента*

имя-аргумента:
  идентификатор **:**

значение-аргумента:
  выражение
  ref *ссылка-на-переменную*
  out *ссылка-на-переменную*

первичное-выражение:
    первичное-выражение-без-создания массива
    выражение-создания-массива

первичное-выражение-без-создания-массива:
    литерал
    простое-имя
    выражение-в-скобках
    доступ-к-элементу
    выражение-вызова
    доступ-к-элементу-массива
    this-доступ
    base-доступ
    пост-инкрементное-выражение
    пост-декрементное-выражение
    выражение-создания-объекта
    выражение-создания-делегата
    выражение-создания-анонимного-объекта
    выражение-typeof
    выражение-checked
    выражение-unchecked
    выражение-значения-по-умолчанию
    выражение-анонимного-метода

простое-имя:
    идентификатор *список-аргументов-типов*$_{opt}$

выражение-в-скобках:
    (*выражение* **)**

доступ-к-элементу:
    первичное-выражение **.** *идентификатор* *список-аргументов-типов*$_{opt}$
    предопределенный-тип **.** *идентификатор* *список-аргументов-типов*$_{opt}$
    уточненный-псевдоним-элемента **.** *идентификатор*
        список-аргументов-типов$_{opt}$

предопределенный-тип: один из

bool	**byte**	**char**	**decimal**	**double**	**float**	**int**	**long**
object	**sbyte**	**short**	**string**	**uint**	**ulong**	**ushort**	

выражение-вызова:
    первичное-выражение **(** *список-аргументов*$_{opt}$ **)**

доступ-к-элементу-массива:
    первичное-выражение-без-создания-массива **[** *список-аргументов* **]**

this-доступ:
    this

base-доступ:
    base **.** *идентификатор*
    base **[** *список-аргументов* **]**

пост-инкрементное-выражение:
    первичное-выражение **++**

пост-декрементное-выражение:
первичное-выражение **--**

выражение-создания-объекта:
new *тип* **(** *список-аргументов*$_{opt}$ **)** *инициализатор-объекта-или-коллекции*$_{opt}$
new *тип* *инициализатор-объекта-или-коллекции*

инициализатор-объекта-или-коллекции:
инициализатор-объекта
инициализатор-коллекции

инициализатор-объекта:
{ *список-инициализаторов-элементов*$_{opt}$ **}**
{ *список-инициализаторов-элементов* **,** **}**

список-инициализаторов-элементов:
инициализатор-элемента
список-инициализаторов-элементов **,** *инициализатор-элемента*

инициализатор-элемента:
идентификатор **=** *значение-инициализатора*

значение-инициализатора:
выражение
инициализатор-объекта-или-коллекции

инициализатор-коллекции:
{ *список-инициализаторов-элементов* **}**
{ *список-инициализаторов-элементов* **,** **}**

список-инициализаторов-элементов:
инициализатор-элемента
список-инициализаторов-элементов **,** *инициализатор-элемента*

инициализатор-элемента:
выражение-без-присваивания
{ *список-выражений* **}**

список-выражений:
выражение
список-выражений **,** *выражение*

выражение-создания-массива:
new *тип-не-массива* **[** *список-выражений* **]** *размерности*$_{opt}$
инициализатор-массива$_{opt}$
new *тип-массива* *инициализатор-массива*
new *размерность* *инициализатор-массива*

выражение-создания-делегата:
new *тип-делегата* **(** *выражение* **)**

выражение-создания-анонимного-объекта:
new *инициализатор-анонимного-объекта*

инициализатор-анонимного-объекта:
{ *список-объявлений элементов*$_{opt}$ **}**
{ *список-объявлений-элементов* **,** **}**

список-объявлений-элементов:
  объявление-элемента
  список-объявлений-элементов **,** *объявление-элемента*

объявление-элемента:
  простое-имя
  доступ-к-элементу
  base-доступ
  идентификатор **=** *выражение*

выражение-typeof:
  typeof **(** *тип* **)**
  typeof **(** *имя-неограниченного-типа* **)**
  typeof **(** **void** **)**

имя-неограниченного-типа:
  идентификатор *описатель-обобщенной-размерности*$_{opt}$
  идентификатор **::** *идентификатор* *описатель-обобщенной-размерности*$_{opt}$
  имя-неограниченного-типа **.** *идентификатор*
    описатель-обобщенной-размерности$_{opt}$

описатель-обобщенной-размерности:
  < *запятые*$_{opt}$ **>**

запятые:
  ,
  запятые **,**

выражение-checked:
  checked **(** *выражение* **)**

выражение-unchecked:
  unchecked **(** *выражение* **)**

выражение-значения-по-умолчанию:
  default **(** *тип* **)**

унарное-выражение:
  первичное-выражение
  + *унарное-выражение*
  - *унарное-выражение*
  ! *унарное-выражение*
  ~ *унарное-выражение*
  пре-инкрементное-выражение
  пре-декрементное-выражение
  выражение-приведения

пре-инкрементное-выражение:
  ++ *унарное-выражение*

пре-декрементное-выражение:
  -- *унарное-выражение*

выражение-приведения:
  (*тип* **)** *унарное-выражение*

выражение-умножения:
    унарное-выражение
    выражение-умножения ***** *унарное-выражение*
    выражение-умножения **/** *унарное-выражение*
    выражение-умножения **%** *унарное-выражение*

выражение-сложения:
    выражение-умножения
    выражение-сложения **+** *выражение-умножения*
    выражение-сложения **-** *выражение-умножения*

выражение-сдвига:
    выражение-сложения
    выражение-сдвига **<<** *выражение-сложения*
    выражение-сдвига *сдвиг-вправо* *выражение-сложения*

выражение-отношения:
    выражение-сдвига
    выражение-отношения **<** *выражение-сдвига*
    выражение-отношения **>** *выражение-сдвига*
    выражение-отношения **<=** *выражение-сдвига*
    выражение-отношения **>=** *выражение-сдвига*
    выражение-отношения **is** *тип*
    выражение-отношения **as** *тип*

выражение-равенства:
    выражение-отношения
    выражение-равенства **==** *выражение-отношения*
    выражение-равенства **!=** *выражение-отношения*

выражение-и:
    выражение-равенства
    выражение-и **&** *выражение-равенства*

выражение-исключающее-или:
    выражение-и
    выражение-исключающее-или **^** *выражение-и*

выражение-включающее-или:
    выражение-исключающее-или
    выражение-включающее-или **|** *выражение-исключающее-или*

условное-выражение-и
    выражение-включающее-или
    условное-выражение-и **&&** *выражение-включающее-или*

условное-выражение-или:
    условное-выражение-и
    условное-выражение-или **||** *условное-выражение-и*

выражение-объединения–с-нулем:
    условное-выражение-или
    условное-выражение-или **??** *выражение-объединения–с-нулем*

лямбда-выражение:
    сигнатура-анонимной-функции **=>** *тело-анонимной функции*

выражение-анонимного-метода:
  delegate *сигнатура-явной-анонимной-функции*$_{opt}$ *блок*

сигнатура-анонимной-функции:
  сигнатура-явной-анонимной-функции
  сигнатура-неявной-анонимной-функции

сигнатура-явной-анонимной-функции
  (*список-параметров-явной-анонимной-функции*$_{opt}$ **)**

список-параметров-явной-анонимной-функции:
  параметр-явной-анонимной-функции
  список-параметров-явной-анонимной-функции **,** *параметр-явной-анонимной-функции*

параметр-явной-анонимной-функции:
  модификатор-параметра-анонимной-функции$_{opt}$ *тип* *идентификатор*

модификатор-параметра-анонимной-функции:
  ref
  out

сигнатура-неявной-анонимной-функции:
  (*список-параметров-неявной-анонимной-функции*$_{opt}$ **)**
  параметр-неявной-анонимной-функции

список-параметров-неявной-анонимной-функции:
  параметр-неявной-анонимной-функции
  список-параметров-неявной-анонимной-функции **,** *параметр-неявной-анонимной-функции*

параметр-неявной-анонимной-функции:
  иденитификатор

тело-анонимной-функции:
  выражение
  блок

выражение-запроса:
  конструкция-from *тело-запроса*

конструкция-from
  from *тип*$_{opt}$ *идентификатор* **in** *выражение*

тело-запроса:
  конструкции-тела-запроса$_{opt}$ *конструкция-select-или-group* *продолжение-запроса*$_{opt}$

конструкции-тела-запроса:
  конструкция-тела-запроса
  конструкции-тела-запроса *конструкция-тела-запроса*

конструкция-тела-запроса:
  конструкция-from
  конструкция-let
  конструкция-where
  конструкция-join
  конструкция-join-into
  конструкция-orderby

конструкция-let
    let *идентификатор* **=** *выражение*

конструкция-where:
    where *булевское-выражение*

конструкция-join:
    join *тип*$_{opt}$ *идентификатор* **in** *выражение* **on** *выражение* **equals** *выражение*

конструкция-join-into
    join *тип*$_{opt}$ *идентификатор* **in** *выражение* **on** *выражение* **equals** *выражение*
        into *идентификатор*

конструкция-orderby:
    orderby *упорядочения*

упорядочения:
    упорядочения **,** *упорядочение*

упорядочение:
    выражение *порядок-сортировки*$_{opt}$

порядок-сортировки:
    ascending
    descending

конструкция select-или-group:
    конструкция-select
    конструкция-group

конструкция-select:
    select *выражение*

конструкция-group:
    group *выражение* **by** *выражение*

продолжение-запроса:
    into *идентификатор* *тело-запроса*

присваивание:
    унарное-выражение *операция-присваивания* *выражение*

операция-присваивания:
    =
    +=
    -=
    ***=**
    /=
    %=
    &=
    |=
    ^=
    <<=
    присваивание-со-сдвигом-вправо

выражение
  выражение-без-присваивания
  присваивание

выражение-без-присваивания:
  условное-выражение
  лямбда-выражение
  выражение-запроса

константное-выражение:
  выражение

булевское-выражение:
  выражение

Б.2.5. Операторы

оператор:
  помеченный-оператор
  оператор-объявления
  вложенный-оператор

вложенный-оператор:
  блок
  пустой-оператор
  оператор-выражение
  оператор-выбора
  оператор-цикла
  оператор-перехода
  оператор-try
  оператор-checked
  оператор-unchecked
  оператор-lock
  оператор-using
  оператор-yield

блок:
  { *список операторов*$_{opt}$ **}**

список-операторов:
  оператор
  список-операторов *оператор*

пустой-оператор:
  ;

помеченный-оператор:
  идентификатор **:** *оператор*

оператор-объявления:
  объявление-локальной-переменной **;**
  объявление-локальной-константы **;**

объявление-локальной-переменной:
  тип-локальной-переменной *описатели-локальной-переменной*

тип-локальной-переменной :
    тип
    var

описатели-локальной-переменной:
    описатель-локальной-переменной
    описатели-локальной-переменной **,** *описатель-локальной-переменной*

описатель-локальной-переменной:
    идентификатор

идентификатор **=** *инициализатор-локальной-переменной*

инициализатор-локальной-переменной:
    выражение
    инициализатор-массива

объявление-локальной-константы:
    const *тип* *описатели-констант*

описатели-констант:
    описатель-константы
    описатели-констант **,** *описатель-константы*

описатель-константы:
    идентификатор **=** *константное-выражение*

оператор-выражение:
    выражение-оператора **;**

выражение-оператора:
    вызов-метода
    создание-объекта
    присваивание
    пост-инкрементное-выражение
    пост-декрементное-выражение
    пре-инкрементное-выражение
    пре-декрементное-выражение

операторы-выбора:
    оператор-if
    оператор-switch

оператор-if:
    if **(** *логическое-выражение* **)** *вложенный-оператор*
    if **(** *логическое-выражение* **)** *вложенный-оператор* **else** *вложенный-оператор*

оператор-switch:
    switch **(** *выражение* **)** *switch-блок*

switch-блок:
    { *switch-секции*$_{opt}$ **}**

switch-секции:
    switch-секция
    switch-секции *switch-секция*

switch-секция:
  switch-метки список-операторов

switch-метки:
  switch-метка
  switch-метки switch-метка

switch-метка:
  case *константное-выражение* **:**
  default :

оператор-цикла:
  оператор-while
  оператор-do
  оператор-for
  оператор-foreach

оператор-while
  while (*логическое-выражение* **)** *вложенный-оператор*

оператор-do:
  do *вложенный-оператор* **while (** *логическое-выражение* **) ;**

оператор-for
  for (*инициализатор-for*$_{opt}$ **;** *условие-for*$_{opt}$ **;** *итератор-for*$_{opt}$ **)**
    вложенный-оператор

инициализатор-for:
  объявление-локальной-переменной
  список-операторов-выражений

условие-for:
  логическое-выражение

итератор-for:
  список-операторов-выражений

список-операторов-выражений
  оператор-выражение
  список-операторов-выражений **,** *оператор-выражение*

оператор-foreach:
  foreach (*тип-локальной-переменной* *идентификатор* **in** *выражение* **)**
  вложенный-оператор

оператор-перехода:
  оператор-break
  оператор-continue
  оператор-goto
  оператор-return
  оператор-throw

оператор-break:
  break ;

оператор-continue:
continue **;**

оператор-goto:
goto *идентификатор* **;**
goto case *константное-выражение* **;**
goto default **;**

оператор-return:
return *выражение*$_{opt}$ **;**

оператор-throw:
throw *выражение*$_{opt}$ **;**

оператор-try:
try *блок* *catch-блоки*
try *блок* *finally-блок*
try *блок* *catch-блоки* *finally-блок*

catch-блоки:
специальные-catch-блоки *общий-catch-блок*$_{opt}$
специальные-catch-блоки$_{opt}$ *общий-catch-блок*

специальные-catch-блоки
специальный-catch-блок
специальные-catch-блоки *специальный-catch-блок*

специальный-catch-блок:
catch **(** *тип-класса* *идентификатор*$_{opt}$ **)** *блок*

общий-catch-блок:
catch *блок*

finally-блок:
finally *блок*

оператор-checked:
checked *блок*

оператор-unchecked:
unchecked *блок*

оператор-lock
lock **(** *выражение* **)** *вложенный-оператор*

оператор-using
using **(** *получение-ресурса* **)** *вложенный-оператор*

получение-ресурса:
объявление-локальной-переменной
выражение

оператор-yield:
yield **return** *выражение* **;**
yield **break** **;**

Б.2.6. Пространства имен

единица-компиляции:
  директивы-внешнего-псевдонима$_{opt}$ *директивы-using*$_{opt}$ *глобальные-атрибуты*$_{opt}$
    объявления-элементов-пространства-имен$_{opt}$

объявление-пространства-имен:
  namespace *уточненное-имя* *тело-пространства-имен* $_{opt}$ **;** $_{opt}$

уточненное-имя:
  идентификатор
  уточненное-имя **.** *идентификатор*

тело-пространства-имен:
  { *директивы-внешнего-псевдонима*$_{opt}$ *директивы-using*$_{opt}$
    объявления-элементов-пространства-имен $_{opt}$ **}**

директивы-внешнего-псевдонима:
  директива-внешнего-псевдонима
  директивы-внешнего-псевдонима *директива-внешнего-псевдонима*

директива-внешнего-псевдонима:
  extern alias *идентификатор* **;**

директивы-using:
  директива-using
  директивы-using *директива-using*

директива-using:
  using-директива-псевдонима
  using-директива-пространства-имен

using-директива-псевдонима:
  using *идентификатор* **=** *имя- пространства-имен-или-типа* **;**

using-директива-пространства-имен:
  using *имя-пространства-имен;*

объявление-элементов-пространства-имен:
  объявление-элемента-пространства-имен
  объявление-элементов-пространства-имен *объявление-элемента-пространства-имен*

объявление-элемента-пространства-имен:
  объявление-пространства-имен
  объявление-типа

объявление-типа:
  объявление-класса
  объявление-структуры
  объявление-интерфейса
  объявление-перечисления
  объявление-делегата

уточненный-псевдоним:
  идентификатор *::* *идентификатор* *список-аргументов-типов*$_{opt}$

Б.2.7. Классы

объявление-класса:
атрибуты$_{opt}$ *модификаторы-класса*$_{opt}$ **partial**$_{opt}$ **class** *идентификатор*
список-параметров-типов$_{opt}$ *базовый-класс*$_{opt}$
ограничения-на-параметры-типы$_{op}$ *тело-класса* **;**$_{opt}$

модификаторы-класса:
модификатор-класса
модификаторы-класса *модификатор-класса*

модификатор-класса:
new
public
protected
internal
private
abstract
sealed
static

список-параметров-типа:
< *параметры-типа* **>**

параметры-типы:
атрибуты$_{opt}$ *параметр-тип*
параметры-типы **,** *атрибуты*$_{opt}$ *параметр-тип*

параметр-тип:
идентификатор

базовый-класс:
: *класс*
: *список-интерфейсов*
: *класс* **,** *список-интерфейсов*

список-интерфейсов:
интерфейс
список-интерфейсов **,** *интерфейс*

ограничения-на-параметры-типы:
ограничение-на-параметр-тип
ограничения-на-параметры-типы *ограничение-на-параметр-тип*

ограничение-на-параметр-тип:
where *параметр-тип* **:** *список-ограничений-на-параметры-типы*

список-ограничений-на-параметры-типы:
первичное-ограничение
вторичное-ограничение
ограничение-конструктора
первичное-ограничение **,** *вторичные-ограничения*
первичное-ограничение **,** *ограничение-конструктора*
вторичные-ограничения **,** *ограничение-конструктора*
первичное-ограничение **,** *вторичные-ограничения* **,**
ограничение-конструктора

первичное-ограничение:
    класс
    class
    struct

вторичные-ограничения:
    интерфейс
    параметр-тип
    вторичные-ограничения **,** *интерфейс*
    вторичные-ограничения **,** *параметр-тип*

ограничение-конструктора:
    new ( )

тело-класса:
    { *объявления-элементов-класса*$_{opt}$ **}**

объявления-элементов-класса:
    объявление-элемента-класса
    объявления-элементов-класса объявление-элемента-класса

объявление-элемента-класса:
    объявление-константы
    объявление-поля
    объявление-метода
    объявление-свойства
    объявление-события
    объявление-индексатора
    объявление-операции
    объявление-конструктора
    объявление-деструктора
    объявление-статического-конструктора
    объявление-типа

объявление-константы:
    атрибуты$_{opt}$ *модификаторы-константы*$_{opt}$ **const** *тип*
        описатели-константы **;**

модификаторы-константы:
    модификатор-константы
    модификаторы-константы модификатор-константы

модификатор-константы:
    new
    public
    protected
    internal
    private

описатели-константы:
    описатель-константы
    описатели-константы **,** *описатель-константы*

описатель-константы:
    идентификатор **=** *константное-выражение*

объявление-поля:
атрибуты$_{opt}$ *модификаторы-поля*$_{opt}$ *тип* *описатели-переменных* **;**

модификаторы-поля:
модификатор-поля
модификаторы-поля *модификатор-поля*

модификатор-поля:
new
public
protected
internal
private
static
readonly
volatile

описатели-переменных:
описатель-переменной
описатели-переменных **,** *описатель-переменной*

описатель-переменной:
идентификатор
идентификатор **=** *инициализатор-переменной*

инициализатор-переменной:
выражение
инициализатор-массива

объявление-метода:
заголовок-метода *тело-метода*

заголовок-метода:
атрибуты$_{opt}$ *модификаторы-метода*$_{opt}$ **partial**$_{opt}$
тип-возвращаемого-значения *имя-элемента* *список-параметров-типов*$_{opt}$
(*список-формальных-параметров*$_{opt}$ **)**
ограничения-на-параметры-типы$_{opt}$

модификаторы-метода:
модификатор-метода
модификаторы-метода *модификатор-метода*

модификатор-метода:
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

тип-возвращаемого-значения:
    тип
    void

имя-элемента:
    идентификатор
    интерфейс . *идентификатор*

тело-метода:
    блок
    ;

список-формальных-параметров:
    фиксированные-параметры
    фиксированные-параметры , *параметр-массив*
    параметр-массив

фиксированные-параметры:
    фиксированный-параметр
    фиксированные-параметры , *фиксированный-параметр*

фиксированный-параметр:
    атрибуты$_{opt}$ *модификатор-параметра*$_{opt}$ *тип* *идентификатор*
        аргумент-по-умолчанию$_{opt}$

аргумент-по-умолчанию:
    = *выражение*

модификатор-параметра:
    ref
    out
    this

параметр-массив:
    атрибуты$_{opt}$ **params** *тип-массива* *идентификатор*

объявление-свойства:
    атрибуты$_{opt}$ *модификаторы-свойства*$_{opt}$ *тип* *имя-элемента*
        { *объявления-кодов-доступа* **}**

модификаторы-свойства:
    модификатор-свойства
    модификаторы-свойства *модификатор-свойства*

модификатор-свойства:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern

имя-элемента:
    идентификатор
    интерфейс **.** *идентификатор*

объявления-кодов-доступа:
    объявление-кода-получения *объявление-кода-установки*$_{opt}$
    объявление-кода-установки *объявление-кода-получения*$_{opt}$

объявление-кода-получения:
    атрибуты$_{opt}$ *модификатор-кода-доступа*$_{opt}$ **get** *тело-кода-доступа*

объявление-кода-установки:
    атрибуты$_{opt}$ *модификатор-кода-доступа*$_{opt}$ **set** *тело-кода-доступа*

модификатор-кода-доступа:
    protected
    internal
    private
    protected internal
    internal protected

тело-кода-доступа:
    блок
    ;

объявление-события:
    атрибуты$_{opt}$ *модификаторы-события*$_{opt}$ **event** *тип*
        описатели-переменных **;**
    атрибуты$_{opt}$ *модификаторы-события*$_{opt}$ **event** *тип* *имя-элемента*
        { *объявления-кодов-доступа-события* **}**

модификаторы-события:
    модификатор-события
    модификаторы-события *модификатор-события*

модификатор-события:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern

объявления-кодов-доступа-события:
    объявление-кода-доступа-add *объявление-кода-доступа-remove*
    объявление-кода-доступа-remove *объявление-кода-доступа-add*

объявление-кода-доступа-add:
    атрибуты$_{opt}$ **add** *блок*

объявление-кода-доступа-remove:
атрибуты$_{opt}$ **remove** *блок*

объявление-индексатора:
атрибуты$_{opt}$ *модификаторы-индексатора*$_{opt}$ *описатель-индексатора*
{ *объявления-кодов-доступа* **}**

модификаторы-индексатора:
модификатор-индексатора
модификаторы-индексатора *модификатор-индексатора*

модификатор-индексатора:
new
public
protected
internal
private
virtual
sealed
override
abstract
extern

описатель-индексатора:
тип **this** **[** *список-формальных-параметров* **]**
тип *интерфейс* **.** **this** **[** *список-формальных-параметров* **]**

объявление-операции:
атрибуты$_{opt}$ *модификаторы-операции* *описатель-операции*
тело-операции

модификаторы-операции:
модификатор-операции
модификаторы-операции *модификатор-операции*

модификатор-операции:
public
static
extern

описатель-операции:
описатель-унарной-операции
описатель-бинарной-операции
описатель-операции-преобразования

описатель-унарной-операции:
тип **operator** *перегружаемая-унарная-операция*
(*тип* *идентификатор* **)**

перегружаемая-унарная-операция: одна из
+ - ! ~ ++ -- true false

описатель-бинарной-операции:
тип **operator** *перегружаемая-бинарная-операция*
(*тип идентификатор* **,** *тип идентификатор* **)**

перегружаемая-бинарная-операция:
+
-
*****
/
%
&
|
^
<<
сдвиг-вправо
==
!=
>
<
>=
<=

описатель-операции-преобразования:
implicit **operator** *тип* **(** *тип* *идентификатор* **)**
explicit **operator** *тип* **(** *тип* *идентификатор* **)**

тело-операции:
блок
;

объявление-конструктора:
атрибуты$_{opt}$ *модификаторы-конструктора*$_{opt}$ *описатель-конструктора*
тело-конструктора

модификаторы-конструктора:
модификатор-конструктора
модификаторы-конструктора *модификатор-конструктора*

модификатор-конструктора:
public
protected
internal
private
extern

описатель-конструктора:
идентификатор **(** *список-формальных-параметров*$_{opt}$ **)**
инициализатор-конструктора$_{opt}$

инициализатор-конструктора:
: **base** **(** *список-аргументов*$_{opt}$ **)**
: **this** **(** *список-аргументов*$_{opt}$ **)**

тело-конструктора:
блок
;

объявление-статического-конструктора:
атрибуты$_{opt}$ *модификаторы-статического-конструктора* *идентификатор*
(**)** *тело-статического-конструктора*

модификаторы-статического-конструктора:
extern$_{opt}$ **static**
static **extern**$_{opt}$

тело-статического-конструктора:
блок
;

объявление-деструктора:
атрибуты$_{opt}$ **extern**$_{opt}$ **~** *идентификатор* **(** **)** *тело-деструктора*

тело-деструктора:
блок
;

Б.2.8. Структуры

объявление-структуры:
атрибуты$_{opt}$ *модификаторы*$_{opt}$ **partial**$_{opt}$ **struct** *идентификатор*
список-параметров-типа$_{opt}$ *интерфейсы-структуры*$_{opt}$
ограничения-на-параметры-типа$_{opt}$ *тело-структуры* **;**$_{opt}$

модификаторы-структуры:
модификатор-структуры
модификаторы-структуры *модификатор-структуры*

модификатор-структуры:
new
public
protected
internal
private

интерфейсы-структуры:
: список-интерфейсов

тело-структуры:
{ *объявления-элементов-структуры*$_{opt}$ **}**

объявления-элементов-структуры:
объявление-элемента-структуры
объявления-элементов-структуры *объявление-элемента-структуры*

объявление-элемента-структуры:
объявление-константы
объявление-поля
объявление-метода
объявление-свойства
объявление-события
объявление-индексатора
объявление-операции
объявление-конструктора
объявление-статического-конструктора
объявление-типа

Б.2.9. Массивы

массив:
не-массив *спецификаторы-размерности*

не-массив:
тип

спецификаторы-размерности:
спецификатор-размерности
спецификаторы-размерности *спецификатор-размерности*

спецификатор-размерности:
[*разделители-размерности*$_{opt}$ **]**

разделители-размерности:
,
разделители-размерности **,**

инициализатор-массива:
{ *список-инициализаторов-переменных*$_{opt}$**,** **}**
{ *список-инициализаторов-переменных* **,** **}**

список-инициализаторов-переменных:
инициализатор переменной
список-инициализаторов-переменных **,** *инициализатор-переменной*

инициализатор-переменной:
выражение
инициализатор-массива

Б.2.10. Интерфейсы

объявление-интерфейса:
атрибуты$_{opt}$ *модификаторы-интерфейса*$_{opt}$ **partial**$_{opt}$ **interface**
идентификатор *список-параметров-вариантного-типа*$_{opt}$
базовые-интерфейсы$_{opt}$ *ограничения-на-параметры-типа*$_{opt}$
тело-интерфейса **;**$_{opt}$

модификаторы-интерфейса:
модификатор-интерфейса
модификаторы-интерфейса *модификатор-интерфейса*

модификатор-интерфейса:
new
public
protected
internal
private

список-параметров-вариантного-типа:
< *параметры-вариантного-типа* **>**

параметры-вариантного-типа:
атрибуты$_{opt}$ *аннотация-вариантности*$_{opt}$ *параметр-типа*
параметры-вариантного-типа **,** *атрибуты*$_{opt}$ *аннотация-вариантности*$_{opt}$
параметр-типа

аннотация-вариантности:
in
out

базовый-интерфейс:
: *список-типов-интерфейсов*

тело-интерфейса:
{ *объявления-элементов-интерфейса*$_{opt}$ **}**

объявления-элементов-интерфейса:
объявление-элементов-интерфейса
объявления-элементов-интерфейса *объявление-элементов-интерфейса*

объявление-элементов-интерфейса:
объявление-метода-интерфейса
объявление-свойства-интерфейса
объявление-события-интерфейса
объявление-индексатора-интерфейса

объявление-метода-интерфейса:
атрибуты$_{opt}$ **new**$_{opt}$ *тип-возвращаемого-значения* *идентификатор*
список-параметров-типа **(** *список-формальных-параметров*$_{opt}$ **)**
ограничения-на-параметры-типа$_{opt}$ **;**

объявление-свойства-интерфейса:
атрибуты$_{opt}$ **new**$_{opt}$ *тип* *идентификатор* **{** *коды-доступа-интерфейса* **}**

коды-доступа-интерфейса:
атрибуты$_{opt}$ **get** **;**
атрибуты$_{opt}$ **set** **;**
атрибуты$_{opt}$ **get** **;** *атрибуты*$_{opt}$ **set** **;**
атрибуты$_{opt}$ **set** **;** *атрибуты*$_{opt}$ **get** **;**

объявление-события-интерфейса:
атрибуты$_{opt}$ **new**$_{opt}$ **event** *тип* *идентификатор* **;**

объявление-индексатора-интерфейса:
атрибуты$_{opt}$ **new**$_{opt}$ *тип* **this** **[** *список-формальных-параметров* **]**
{ *коды-доступа-интерфейса* **}**

Б.2.11. Перечисления

объявление-перечисления:
атрибуты$_{opt}$ *модификаторы*$_{opt}$ **enum** *имя* *базовый-тип*$_{opt}$ *тело-перечисления* **;**$_{opt}$

базовый-тип:
: *целочисленный-тип*

тело-перечисления:
 { *объявление-элементов-перечисления*$_{opt}$ **}**
 { *объявление-элементов-перечисления*$_{opt}$ **,** **}**

модификаторы-перечисления:
 модификатор-перечисления
 модификаторы-перечисления *модификатор-перечисления*

модификатор-перечисления:
 new
 public
 protected
 internal
 private

объявление-элементов-перечисления:
 объявление-элемента-перечисления
 объявление-элемента-перечисления **,** *объявление-элемента-перечисления*

объявление элементов перечисления:
 атрибуты$_{opt}$ *идентификатор*
 атрибуты$_{opt}$ *идентификатор* **=** *константное-выражение*

Б.2.12. Делегаты

объявление-делегата:
 атрибуты$_{opt}$ *модификаторы-делегата*$_{opt}$ **delegate**
 тип-возвращаемого-значения *идентификатор*
 список-параметров-вариантного-типа$_{opt}$
 (*список-формальных-параметров*$_{opt}$ **)** *ограничения-на-параметры-типа*$_{opt}$ **;**

модификаторы-делегата:
 модификаторы-делегата
 модификаторы-делегата *модификатор-делегата*

модификатор-делегата:
 new
 public
 protected
 internal
 private

Б.2.13. Атрибуты

глобальные-атрибуты:
 секции-глобальных-атрибутов

секции-глобальных-атрибутов:
 секция-глобальных-атрибутов
 секции-глобальных-атрибутов *секция-глобальных-атрибутов*

секция-глобальных-атрибутов:
 [*спецификатор-целевого-объекта-глобального-атрибута* *список-атрибутов* **]**
 [*спецификатор-целевого-объекта-глобального-атрибута* *список-атрибутов* **,** **]**

спецификатор-целевого-объекта-глобального-атрибута
    целевой-объект-глобального-атрибута **:**

целевой-объект-глобального-атрибута:
    assembly
    module

атрибуты:
    секции-атрибутов

секции-атрибутов:
    секция-атрибутов
    секции-атрибутов *секция-атрибутов*

секция-атрибутов:
    [*спецификатор-целевого-объекта-атрибута*$_{opt}$ *список-атрибутов* **]**
    [*спецификатор-целевого-объекта-атрибута*$_{opt}$ *список-атрибутов* **,** **]**

спецификатор-целевого-объекта-атрибута:
    целевой-объект-атрибута **:**

целевой-объект-атрибута:
    field
    event
    method
    param
    property
    return
    type

список-атрибутов:
    атрибут
    список-атрибутов **,** *атрибут*

атрибут:
    имя-атрибута *аргументы-атрибута*$_{opt}$

имя-атрибута:
    имя-типа

аргументы-атрибута:
    (*список-позиционных-аргументов*$_{opt}$ **)**
    (*список-позиционных-аргументов* **,** *список-именованных-аргументов* **)**
    (*список-именованных-аргументов* **)**

список-позиционных-аргументов:
    позиционный-аргумент
    список-позиционных-аргументов **,** *позиционный-аргумент*

позиционный-аргумент:
    имя-аргумента$_{opt}$ *выражение-аргумента-атрибута*

список-именованных-аргументов:
    именованный-аргумент
    список-именованных-аргументов **,** *именованный-аргумент*

именованный-аргумент:
    идентификатор **=** *выражение-аргумента-атрибута*

выражение-аргумента-атрибута:
    выражение

Б.3. Расширения грамматики для небезопасного кода

модификатор-класса:
    …
    unsafe

модификатор-структуры:
    …
    unsafe

модификатор-интерфейса:
    …
    unsafe

модификатор-делегата:
    …
    unsafe

модификатор-поля:
    …
    unsafe

модификатор-метода:
    …
    unsafe

модификатор-свойства:
    …
    unsafe

модификатор-события:
    …
    unsafe

модификатор-индексатора:
    …
    unsafe

модификатор-операции:
    …
    unsafe

модификатор-конструктора:
    …
    unsafe

```
объявление-деструктора:
    атрибуты_opt   extern_opt   unsafe_opt   ~   идентификатор   (  )
            тело-деструктора
    атрибуты_opt   unsafe_opt   extern_opt   ~   идентификатор   (  )
           тело-деструктора

модификаторы-статического-конструктора:
    extern_opt   unsafe_opt   static
    unsafe_opt   extern_opt   static
    extern_opt   static   unsafe_opt
    unsafe_opt   static   extern_opt
    static   extern_opt   unsafe_opt
    static   unsafe_opt   extern_opt

вложенный-оператор:
    …
    оператор-unsafe

оператор-unsafe:
    unsafe   блок

тип:
    …
    тип-указателя

тип-указателя:
    неуправляемый-тип   *
    void  *

неуправляемый-тип:
    тип

первичное-выражение-без-создания-массива:
    …
    доступ-к-элементу-объекта-через-указатель
    доступ-к-элементу-через-указатель
    выражение-sizeof

унарное-выражение:
    …
    выражение-разыменования-указателя
    выражение-получения-адреса

выражение-разыменования-указателя:
    *    унарное-выражение

доступ-к-элементу-объекта-через-указатель:
    первичное-выражение   ->   идентификатор   список-аргументов-типа_opt

доступ-к-элементу-через-указатель:
    первичное-выражение-без-создания-массива   [   выражение   ]

выражение-получения-адреса:
    &   унарное-выражение
```

выражение-sizeof:
sizeof **(** *неуправляемый-тип* **)**

оператор-fixed:
fixed **(** *тип-указателя* *описатели-фиксированного-указателя* **)**
вложенный-оператор

описатели-фиксированного-указателя:
описатель-фиксированного-указателя
описатели-фиксированного-указателя **,** *описатель-фиксированного-указателя*

описатель-фиксированного-указателя:
идентификатор **=** *инициализатор-фиксированного-указателя*

инициализатор-фиксированного-указателя:
& *ссылка-на-переменную*
выражение

объявление-элемента-структуры:
…
объявление-буфера-фиксированного-размера

объявление-буфера-фиксированного-размера:
атрибуты$_{opt}$ *модификаторы-буфера-фиксированного-размера*$_{opt}$ **fixed**
тип-элементов-буфера *описатели-буферов-фиксированного-размера* **;**

модификаторы-буфера-фиксированного-размера:
модификатор-буфера-фиксированного-размера
модификатор-буфера-фиксированного-размера
модификаторы-буфера-фиксированного-размера

модификатор-буфера-фиксированного-размера:
new
public
protected
internal
private
unsafe

тип-элементов-буфера:
тип

описатели-буферов-фиксированного-размера:
описатель-буфера-фиксированного-размера
описатель-буфера-фиксированного-размера **,**
описатели-буферов-фиксированного-размера

описатель-буфера-фиксированного-размера:
идентификатор **[** *константное-выражение* **]**

инициализатор-локальной-переменной:
…
инициализатор-stackalloc

инициализатор-stackalloc:
stackalloc *неуправляемый-тип* **[** *выражение* **]**

Приложение В
Ссылки

IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard 754-1985. Доступно по адресу *http://www.ieee.org*.

ISO/IEC. C++. ANSI/ISO/IEC 14882:1998.

Unicode Consortium. The Unicode Standard, Version 3.0. Addison-Wesley, Reading. Massachusetts, 2000. ISBN 0-201-616335-5.

А. Хейлсберг, М. Торгерсен, С. Вилтамут, П. Голд

Язык программирования C#. Классика Computers Science
4-е издание

Перевел с английского Р. Тетерин

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Научный редактор	*Т. Павловская*
Художественный редактор	*К. Радзевич*
Корректор	*В. Листова*
Верстка	*Б. Файзуллин*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 30.08.11. Формат 70х100/16. Усл. п. л. 63,210. Тираж 1500. Заказ 0000.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C#. 3-е издание

Дж. Рихтер



ISBN: 978-5-459-00297-3

Объем: 976 с.

Выход: ноябрь 2011

В этой книге, выходящей в третьем издании и уже ставшей классическим учебником по программированию, подробно описывается внутреннее устройство и функционирование общеязыковой исполняющей среды (CLR) Microsoft .NET Framework версии 4.0. Написанная признанным экспертом в области программирования Джеффри Рихтером, много лет являющимся консультантом команды разработчиков .NET Framework компании Microsoft, книга научит вас создавать по-настоящему надежные приложения любого вида, в том числе с использованием Microsoft Silverlight, ASP.NET, Windows Presentation Foundation и т. д.

Третье издание полностью обновлено в соответствии со спецификацией платформы .NET Framework 4.0 и принципами многоядерного программирования.

Компьютерные сети
5-е издание

Э. Таненбаум, Д. Уэзеролл



ISBN: 978-5-459-00342-0
Объем: 1040 с.
Выход: октябрь 2011

Перед вами — очередное, пятое издание самой авторитетной книги по современным сетевым технологиям, написанной признанным экспертом в этой области Эндрю Таненбаумом в соавторстве с профессором Вашингтонского университета Дэвидом Уэзероллом. Первая версия этого классического труда появилась на свет в далеком 1980 году, и с тех пор каждое издание книги неизменно становилось бестселлером и использовалось в качестве базового учебника в ведущих технических вузах.
В книге последовательно изложены основные концепции, определяющие современное состояние и тенденции развития компьютерных сетей. Авторы подробнейшим образом объясняют устройство и принципы работы аппаратного и программного обеспечения, рассматривают все аспекты и уровни организации сетей — от физического до уровня прикладных программ. Изложение теоретических принципов дополняется яркими, показательными примерами функционирования Интернета и компьютерных сетей различного типа. Пятое издание полностью переработано с учетом изменений, происшедших в сфере сетевых технологий за последние годы и, в частности, освещает такие аспекты, как беспроводные сети стандарта 802.12 и 802.16, сети 3G, технология RFID, инфраструктура доставки контента CDN, пиринговые сети, потоковое вещание, интернет-телефония и многое другое.

Идеальная разработка ПО. Рецепты лучших программистов

Под ред. Э. Орама, Г. Уилсона



ISBN: 978-5-459-01099-2

Объем: 592 с.

Выход: октябрь 2011

Авторы популярной в IT-сообществе книги «Идеальный код» вновь предлагают вашему вниманию подборку лучших решений от признанных экспертов в области разработки ПО.

Существует много споров о том, какие же инструменты, технологии и практики могут действительно оптимизировать процесс разработки ПО и усовершенствовать конечный продукт. В новой книге под редакцией Энди Орама и Грега Уилсона известные разработчики делятся своим бесценным опытом и мнениями на эту тему. Авторские эссе и статьи посвящены наиболее эффективным методам работы программиста, а также развенчиванию ряда мифов, существующих в программистском сообществе.

Среди авторов книги — такие авторитеты, как Стив Макконнелл, Барри Бэм, Барбара Китченхем и еще 27 известных экспертов в области разработки программного обеспечения.

Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е издание

Р. Лафоре



ISBN: 978-5-459-00292-8
Объем: 704 с.
Выход: июнь 2011

Второе издание одной из самых авторитетных книг по программированию посвящено использованию структур данных и алгоритмов. Алгоритмы — это основа программирования, определяющая, каким образом разрабатываемое программное обеспечение будет использовать структуры данных. На четких и простых программных примерах автор объясняет эту сложную тему, предлагая читателям написать собственные программы и на практике усвоить полученные знания. Рассматриваемые примеры написаны на языке Java, хотя для усвоения материала читателю не обязательно хорошо знать его — достаточно владеть любым языком программирования, например C++. Первая часть книги представляет собой введение в алгоритмизацию и структуры данных, а также содержит изложение основ объектно-ориентированного программирования. Следующие части посвящены различным алгоритмам и структурам данных, рассматриваемым от простого к сложному: сортировка, абстрактные типы данных, связанные списки, рекурсия, древовидные структуры данных, хеширование, пирамиды, графы. Приводятся рекомендации по использованию алгоритмов и выбору той или иной структуры данных в зависимости от поставленной задачи.